Search and Query Framework for Workflows with HPC and AI Models

Christopher D. Rickett, Sreenivas R. Sukumar, Karlon K. West
Hewlett Packard Enterprise

chris.rickett@hpe.com
sreenivas.sukumar@hpe.com
karlon.west@hpe.com

Abstract—Modern computational science workflows increasingly involve complex, interactive, and iterative search through data from simulations of physics-based equations coupled with analytic, predictive, generative, and agentive tasks. Unfortunately, there are no query engines that empower scientists to search through scientific data with AI, analytic and physics-based models like we do searches with SQL query engines on structured data or keyword-search/prompt engines for textual data.
This paper presents an intelligent data search framework that empowers scientists to query scientific data. The framework functionality currently consists of: (i) a query engine and index for keyword, set-theoretic and linear-algebraic search (ii) a repository of domain-specific models, open-source code or pre-trained AI models, and (iii) a query planner for mixed HPC simulations and AI inferences.
We demonstrate this novel search framework on a life-sciences use-case in collaboration with the National Center for Natural Products in Mississippi. We leverage publicly available AI models such as AlphaFold, MolGAN, DTBA as AI surrogates for structure prediction, molecular generation, and binding-affinity prediction along with theoretical HPC codes such as Autodock Vina and domain-specific user-defined functions for protein similarity estimation such as the Smith-Waterman algorithm to search and hypothesize healthy alternatives to existing compounds such as caffeine or zero-calorie sugar. We show how our search framework answers the ”what-is,” ”what-else,” ”what-if,” and ”what-could-be” questions.

Index Terms—graph analytics, AI powered queries, PGAS, parallel programming, pattern search, pattern mining, Cray Graph Engine.

1 Introduction

Computational science stands as a cornerstone of modern scientific discovery, enabling researchers across diverse fields—from materials science and drug discovery to climate modeling and fundamental physics behind the formation of the universe—to simulate complex phenomena governed by physics-based equations. These simulations generate unprecedented volumes of data, offering deep insights into intricate systems. Increasingly, this simulation-driven approach is intertwined with the power of artificial intelligence (AI) and machine learning (ML). AI/ML models excel at uncovering patterns, building predictive surrogates for expensive simulations, generating novel hypotheses (e.g., new molecular structures or material compositions), and even acting as intelligent agents within complex experimental loops within the discovery workflow.

However, this exciting progress reveals a critical bottleneck: how do scientists effectively search, interrogate, and interact with the vast, complex datasets produced by computational workflows with AI? How can scientists leverage the tremendous progress in AI to search through images, videos, point-clouds and other unstructured data formats? The scientific workflow is rarely linear and rigorous. It involves iterative cycles of hypothesis generation, testing through simulation or AI prediction, analysis of results, and refinement of the next query. Scientists need to ask nuanced questions that weave together data from simulation outputs (e.g., time-series of particle positions, stress tensors, protein conformations) with insights from analytical models, predictions from pre-trained AI models (e.g., property prediction, structure classification), and even results from generative models or brief, on-the-fly simulations. Furthermore, for reproducibility and statistical significance critical to scientific rigor scientists need to show irrefutable evidence – i.e. robust to the choice of simulation parameters, the analytical method and AI models etc.

Consider a typical scenario in drug discovery: From an archive of prior simulations, a researcher might want to find all simulations where a molecule exhibited a certain binding potential (from simulation data), cross-reference this with predicted toxicity scores from an AI model, identify structurally similar molecules using domain-specific algorithms, and perhaps even generate novel candidate molecules with potentially improved properties using a generative AI model, all within an integrated exploration process. The search methodology requires the ability to optimally plan and execute thousands of simulations, millions of analytical functions and AI inferences.

Unfortunately, our current data query paradigms fall significantly short of supporting such integrated, interactive, and computationally rich scientific exploration. Relational database systems, queried via SQL, excel at structured, tabular data but struggle with the complex, often

unstructured simulation outputs and lack native support for incorporating computational models directly into queries. Keyword search and modern prompt-based engines are powerful for textual information retrieval but are ill-suited for querying numerical data based on scientific principles, structural similarity, or the outputs of predictive models. Existing scientific data management platforms may store simulation data but often offer limited querying capabilities, typically restricted to metadata searches or predefined analyses. They generally lack the ability to dynamically invoke and integrate physics-based simulations, AI inferences, and custom analytical functions as first-class citizens within the query execution itself.

These disconnects forces scientists into cumbersome, manual workflows, stitching together disparate tools: retrieving data from a database, processing it with separate scripts, feeding results into an AI model via another interface, running a new simulation snippet, and manually collating the findings. This process is inefficient, error-prone, and fundamentally hinders the pace and scope of discovery by making complex, iterative exploration prohibitively difficult.

To address this critical gap, we introduce an intelligent data search framework designed explicitly for the complex demands of modern computational science. Our framework empowers scientists to query vast scientific datasets using a unified approach that seamlessly integrates keyword search, set-theoretic operations, and linear-algebraic methods—essential for handling diverse data types and modern embedding-based AI techniques. Crucially, it goes beyond simple data retrieval by incorporating a repository of computational models—spanning domain-specific algorithms, open-source codes, pre-trained AI models, and connections to HPC simulation tools—and an intelligent query planner. This planner orchestrates complex queries that may require executing both traditional HPC simulations and fast AI inferences as part of the query resolution process.

This paper details the architecture and capabilities of our framework. We demonstrate its practical utility through an image search and life-sciences use case. The life-sciences use-case was developed in collaboration with the National Center for Natural Products Research (NCNPR) in Mississippi. For NCNPR, we leverage the framework to search for and hypothesize potentially healthy alternatives to common compounds like caffeine or zero-calorie sugars. We showcase the framework's ability to integrate publicly available AI models, such as AlphaFold for structure prediction, MolGAN for molecular generation, and models for Drug Target Binding Affinity (DTBA) prediction (used here as efficient AI surrogates), alongside established theoretical HPC codes like Autodock Vina for molecular docking, and domain-specific functions like the Smith-Waterman algorithm for protein similarity assessment. Based on this use case, we illustrate how our search framework enables scientists to move beyond simple data lookup and ask sophisticated scientific questions, effectively addressing the "what-is" (descriptive), "what-else" (similarity/alternative finding), "what-if" (hypothetical simulation/prediction), and "what-could-be" (generative exploration) facets of the scientific discovery process. As seen in section 5.3.1, a query on a 100 billion medical fact graph can perform millions of domain-specific functions, hundreds or thousands of AI inferences and thousands of HPC simulations in approximately 60 seconds on 256 nodes. By bridging the gap between data generation and intelligent exploration, our framework aims to significantly accelerate discovery cycles in computational science.

In doing so, we claim the following contributions: (i) the development and implementation of an intelligent data search platform for scientific data (ii) domain-agnostic demonstration of scientific data search using database like queries that can include linear-algebraic functions and domain-specific codes, (iii) strong scaling feature of the search framework to large data sizes, number of models/functions per query and (iv) experiments that showcase the improvements with our optimal query planner for the mixed workloads.

The remainder of this paper is structured as follows: Section 2 elaborates on the background of the intelligent search framework, and sections 3 and 4 provide details for the query engine, model repository, and query planner. Section 5 presents What-is, What-else, What-if, What-could-be, outlining the specific queries and showcasing the results obtained using the framework. Sections 4, 5 and 7 discuss the implications, limitations, and future directions of this work. Finally, Section 6 provides concluding remarks.

2 Background

The Intelligent Data Search (IDS) platform proposed in this paper is built upon 10+ years of prior work on the Cray Graph Engine (CGE). The first semantic graph database developed at Cray was the Urika-GD database appliance, launched in 2012, which leveraged the Cray XMT2 shared memory machine with high performance indirect addressing, which allowed graphs to be searched efficiently with a multi-threaded programming model for hiding latency. The Urika-GD database was ported to the XC30 architecture in 2015, as described in a previous CUG paper [1]. This was accomplished by making use of the Partitioned Global Address Space (PGAS) programming model and Coarray-C++.

Later efforts were completed to enable CGE to execute optimally across multiple platforms, including HPE Cray EX supercomputers, shared memory machines such as HPE SuperDome Flex, parallel cluster, and even cloud deployments. This work centered around replacing the Coarray-C++ and PGAS libraries used by CGE with versions built on top of MPI and POSIX shared memory to support optimal performance and portability. This work has been described in detail in previous studies [2].

The remainder of this paper will describe how new functionality, components and application programming interfaces (APIs) were built on top of CGE to create IDS.

3 Enabling HPC and AI Powered Queries

Our proposed technology solution involves the design and implementation of a massively parallel data search platform that (a) stores, handles, hosts, and processes multi-modal data represented as knowledge graphs, (b) provides interactive query and semantic-traversal capabilities

for data-driven discovery, (c) accelerates domain-specific user-defined functions, as well as vertex-centric and whole-graph algorithms like PageRank for graph-theoretic connectivity and relevance analysis, and (d) executes workflows of queries across multiple datasets to generate hypotheses in seconds rather than months. The architecture of IDS aims to ease user efforts for creating query workflows that mix knowledge graphs with domain-specific user defined functions (UDF), AI operations and HPC-style simulations. Figure 1 illustrates the architecture of our solution.

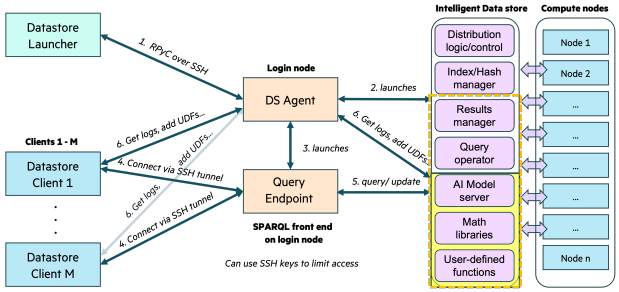


Fig. 1. Architecture of HPE's Intelligent Data Store

Some of the key components and the common functions they perform are:

- DataStore object, used to perform user operations such as launch, opening the query/update endpoint and tear down
- DataStoreClient object is used to submit queries/updates, fetch logs, and add new user codes
- DSAgent object, which exists on the compute hardware, typically a login node, and works with the DataStore object to launch/tear down, and with the DataStoreClient to create connections to the query endpoint, retrieve logs, and import new user codes
- Intelligent Data Store backend, which is built on top of CGE, is the massively parallel graph engine executing on the compute node(s)

The following sections describe in more detail the components of IDS and how they are used to enable queries that mix traditional graph operations, domain specific UDFs, AI powered UDFs and HPC style simulations.

3.1 Python Client API

The design of IDS is intended to make it easier for data scientists to leverage it from Python since many AI codes are implemented using Python. IDS uses a Python client API to enable users to do everything necessary to utilize IDS from a Python script or Jupyter notebook, including: launch, query, ingest data, add new AI functions, and tear down. Further, IDS builds upon the previous work to containerize CGE [2] by including the IDS Python client code and other bits necessary for executing optimally on the given hardware right into the container. This makes it easy for users to start with IDS on their laptop and then move to a larger system by using the same container. The client code for launching, etc, utilize the RPyC [3] package with SSH tunnels to securely access remote compute hosts that the user specifies for hosting IDS.

3.2 Python Code Import

IDS enables users to easily import their own Python code using a function named *import_python*, either by providing the name of a single Python file or an entire directory. IDS will copy the provided code to the host executing IDS by utilizing the API exposed by the DSAgent to the DataStore.

3.3 User Load Function

To enable users to execute their own code for ingesting/-generating data to load, which could include the use of pretrained AI models, the IDS API includes a *load* function. This allows users to specify a program to execute and any arguments and it is expected to return the triples in the file name passed to the user program. Each image involved in hosting an IDS instance participates in a *load* operation, which gives users an easy way to execute an ingest program in an embarrassingly parallel manner for ingesting large datasets.

3.4 Python User Defined Functions

Previous work on CGE enabled users to define their own functions that could be compiled into a shared object that CGE would load at launch time [4]. This was intended primarily for C/C++ programs, such as those used in the previous work for finding potential drugs that could be repurposed against Covid-19 [4]. These UDFs are referred to as static since they cannot be changed once IDS has started.

To better enable users to continually incorporate new AI models or domain specific functions, a new API was added for the dynamic addition of Python UDFs. This is very similar to the existing API for static functions, which created a method named *arq:user_func* to call out to a predefined user entry function and pass arguments from a query to the user function [4]. The new API for dynamically loading Python modules and functions is nearly identical, with the query entry being: *arq:py_user_func*, and two required arguments with the first being the module name to load and the second the name of the function to execute.

Given that Python modules may often be time consuming to load, IDS maintains a cache of the loaded user modules so the overhead is only incurred the first time the module loads. However, to enable users to continually make changes to their code and use it in running queries, IDS has a special function (*cge_py_reload_run*) that users can call to force IDS to reload the module. If users put this function name before the module name argument to *arq:py_user_func*, IDS will reload the module listed before executing the given function and will cache the updated version of the module.

The addition of these key features to IDS make it far more flexible for users in the types of queries they can execute and even the input data that can be processed, either at load time or dynamically within queries. However, UDFs that incorporate pretrained AI models or simulation style codes can vary significantly in compute requirements, which can make it extremely difficult for users to plan the workflows for optimal execution. In the following section, Query Planning for HPC and AI, we will discuss optimizations made to IDS to help address the issue of efficiently scheduling query operations with UDFs.

4 Query Planning for HPC and AI

Key features in IDS are intended to improve the efficiency of executing query workflows with multiple UDFs, which can be AI powered, domain specific computational kernels, and simulation style functions. While users can do some of the query optimizations based upon knowledge about the UDFs, this can quickly become too burdensome as the number of UDFs per query grows. Given the rate at which new AI models are published to sites such as HuggingFace [5] and the ability for users to easily import new codes dynamically into IDS, it is easy to imagine users interactively searching data to test new models with little knowledge of the performance implications. Further, as users refine their subgraphs through their searches, the behavior of UDFs could easily change due to differences in the data. These reasons emphasize why it is essential for IDS to be able to dynamically optimize UDF powered queries to improve performance. The sections below describe how IDS performs these query optimizations.

4.1 Profiling UDF Operations

A key requirement for enabling effective query planning and optimization for queries involving UDFs is for IDS to gather and maintain profiling information about each UDF. For profiling, statically linked and dynamically loaded UDFs are treated the same and profiling is gathered for both. To enable this, the statically linked UDFs are now required to have the first parameter be a unique name for the UDF that can be used as a key in the table IDS creates to track profiling information. As discussed above, the dynamically loaded UDFs are already required to use the first two parameters to provide the name of the Python module to load and the name of the function to execute, and these two are used in combination to create a key in the IDS profiling table.

Each image will independently collect the following profiling data for every UDF executed:

- Number of times a UDF has been executed
- Total amount of time spent executing the UDF
- Number of times the UDF resulted in the rejection of an expression

The first time a UDF is executed, a new entry into the profiling table will be created and the data will be updated each subsequent time the UDF is executed. This data is updated throughout the life time of a running IDS instance, which can be essential for optimizing user queries dynamically over time as users may search different subsets of the data that may have varying compute requirements. Further, this profiling information can vary for each compute image, especially if the IDS images are executing in a heterogeneous environment, so it is important for images to maintain their own profiling statistics to better enable the optimizations that will be discussed in the following sections.

4.2 Solution Rebalancing

One way in which the UDF profiling information is utilized is during solution rebalancing performed at the beginning of FILTER operations. It is common for IDS to rebalance solutions across images between operations (e.g., scan/join/merge) by aiming to have approximately the same number of solutions to process on each image. However, this may not always provide the best balance when considering evaluation of UDF expressions if some images are faster than others. Images can vary in how quickly they can complete a UDF for multiple reasons, such as different hardware available on the node where the image resides or differences in the subgraph each image stores in their shard of the overall graph. For these reasons, enabling rebalancing to consider the rate at which images can execute any UDFs involved in a query could result in lower overall query times.

Within IDS, the set of expressions to evaluate as part of an ORDER or FILTER operation are represented as expression trees. This enables FILTER to evaluate the expressions and perform short-circuiting of conditionals when possible to reduce execution time by avoiding unnecessary evaluations. As part of the FILTER operation, if the expression tree contains UDFs, each image will traverse the expression tree to estimate the time required for evaluating each expression, leveraging the UDF profiling information gathered previously. If no information is available for a given UDF then the estimate is 0 initially.

Once each image determines the estimated time for evaluating the expression tree for a single solution, all images will exchange their estimates and use them to determine the ratio between the slowest image and all other images. If all images have a reasonably similar throughput, say within 20% of the slowest image, the solutions are rebalanced in the typical, straightforward manner of trying to have each image evaluate the same number of solutions. However, in situations where some images are considerably faster than others at evaluating UDFs, the rebalancing of solutions is done in the following manner:

- Use slowest image to determine minimum number of operations per second and calculate ratio of each image relative to the slowest
- Redistribute solutions to evaluate based upon ratios. This is done by summing up the ratios across images and dividing the intermediate solutions to evaluate by this number. This is multiplied by the ratio for each image to determine how many solutions to assign to a given image. For example:
  - 1.4 million intermediate solutions
  - 900 images
    - 500 images have 100/second
    - 300 images have 200/second
    - 100 images have 300/second
  - total chunks to split input into:
    - 500 + (300 * 2) + (100 * 3) = 1400
  - 1.4M / 1.4K = 10K intermediate solutions per chunk
    - 500 slowest images each get 10K solutions to evaluate
    - 300 medium images each get 20K solutions to evaluate

    - 100 fastest images each get 30K solutions to evaluate

- Evaluation of the UDF(s) is independent across images, so the overall time should be equal to the time of the slowest image to evaluate their solutions. In the rebalancing mentioned above, this could be ∼10K/100 = 100 seconds. If a simple rebalancing of equal solutions across images was done instead, then the estimated time would be ∼14K / 100 per second = 140 seconds.

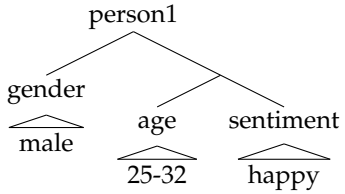
As described above, the utilization of the UDF profiling information for rebalancing intermediate solutions during FILTER expression evaluations can be much more optimal than a simple approach. In section 5.2.1 below, we will demonstrate how this rebalancing can improve overall query performance, even on a homogeneous system.

4.3 Optimizing AI Powered Expressions

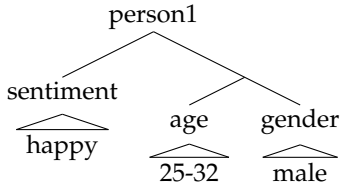
Before executing expressions that contain UDF calls as part of a FILTER operation, each image in IDS uses the profiling information to estimate the time required for each UDF within the expression tree. With this information, each image will traverse the expression tree to create sets of chained conditional operators. If a set of chained conditional operators exists, and at least one of the sub-expressions calls a UDF that profiling information is available for, the images will reorder the expressions in the chain such that the trees are processed in ascending order based on the estimated evaluation time. If there are relative ties in computational time for two UDFs, the ties are broken using the profiling information regarding how often a given UDF results in a solution being eliminated. If that information is available, the UDF that is expected to eliminate more solutions is ordered first, with the intention that it will reduce the number of evaluations executed for each intermediate solution.

This reordering is currently done on an image basis so each image is independent of others in deciding how to order the operations for their subset of the solutions being evaluated. This is possible since the expression evaluation is independent and it is not until all images finish the evaluations on their subset that the images will sync solutions globally. This allows images to have different orderings to best suit the compute resources they are executing upon, which may differ (e.g., different GPUs or mix of CPUs/GPUs).

For example, if our expression tree consisted of testing that a given person in an image was a Male, between the ages of 25-32, and was happy, we might have an expression tree similar to the following:



However, if our UDF profiling showed us that the sentiment analysis was the fastest UDF and the gender prediction the slowest, the reordering optimizations would change the expression tree to the following:



4.4 Query Optimization Summary

In this section, we discussed key optimizations that were added to IDS which leverage profiling statistics for UDFs to dynamically improve query performance. Since IDS enables users to easily add new UDFs that could leverage different AI models, or users may search different subgraphs of the data, the computational requirements of queries with UDFs could easily change over time. It would be nearly impossible to expect users to track the performance of multiple AI models and other UDFs within their queries in order to best optimize query performance. Therefore, the automatic query optimizations discussed in the preceding sections are an essential part of IDS, enabling users to efficiently mix knowledge graph style queries, domain-specific UDFs and AI powered queries as part of an HPC and AI workflow.

In the following section, we will demonstrate how these optimizations play a significant role in improving query performance for multiple use cases that leverage AI and domain-specific UDFs.

5 What-is, What-else, What-if, What-could-be

The following section demonstrates how IDS can be used to power search based workflows that include traditional knowledge graph searches, AI powered queries for probabilistic searches, domain-specific UDFs and finally, queries that combine all of these as well as HPC-style simulations.

All use cases demonstrated were hosted on an HPE Cray EX 1000-node internal benchmarking system using the Slingshot interconnect. The compute nodes use AMD EPYC Rome processors (dual-socket nodes, 64-cores per socket, 512 GB RAM per node). The attached Lustre file system is a Clusterstor system with 12 OSTs providing 2 PB of storage. IDS was built into an Ubuntu 22.04 based Apptainer [6] container that includes OpenMPI 5 [7], libfabric 2.1 [8], and libcxi shs-12.0 [9].

5.1 Knowledge Graphs

First, a trillion triples dataset. We created an instance of the well-known Lehigh University benchmark (LUBM) [10], an artificial test set that describes the structure of universities with departments, courses, and students. Despite the rather simple ontology, it has turned out to be very similar to many realistic data sets, and the availability of scalable generators has made it a standard benchmark for graph databases. We used the multi-threaded `lubm-uba` generator [11] which directly produces N-triple format, and can consolidate output into multiple files to keep instance generation time manageable. We used a scale of 5,500,000 universities (5500K), which yields an initial data set of around 0.75 trillion triples, and

inferencing was performed using the rules provided by [12] to produce another 0.25 trillion. In the end, the generated triples totaled ∼128 TB on disk and there was 1.002 trillion triples in the graph.

The performance of the LUBM queries with IDS can also be compared to the same queries executed for Cambridge Semantics [13] for their trillion triples benchmarks as well as CGE 2018 [14]. For Cambridge Semantics with LUBM 4400K on 200 nodes of the Google Compute Cloud, query 9 took 268.51 seconds and the total query time for all 14 LUBM queries was 839.75 seconds [13]. For CGE 2018 running on an internal XC40 system with 850 nodes and 16 images per node, query 9 took 30.8 seconds and the query time for all 14 LUBM queries took 96.3 seconds. With IDS and LUBM 5500K as shown in Figure 3, query 9 took 12.8 seconds on 800 nodes and 32 images per node (39.5 seconds on 200 nodes) and as seen in Figure 2, the total query time for 12 of the 14 LUBM queries was 77.5 seconds. We excluded LUBM queries 6 and 14 since they perform a simple scan and are extremely fast, but generate a very large result set.

After inferencing, the LUBM 5500K dataset contains 1,002,176,290,661 unique quads. For CGE 2018 on 850 nodes at 16 images per node, this equates to querying 73.69 million quads per image. For IDS 2025, on 800 nodes with 32 images per node, this equates to 39.15 million quads per image. The query 9 and total LUBM query times with IDS are nearly an order of magnitude faster than the Cambridge Semantics benchmark and IDS query 9 is almost three times faster than the CGE 2018 query 9 time, clearly demonstrating the superior performance and scaling capabilities of IDS that enable users to interactively analyze very large datasets.

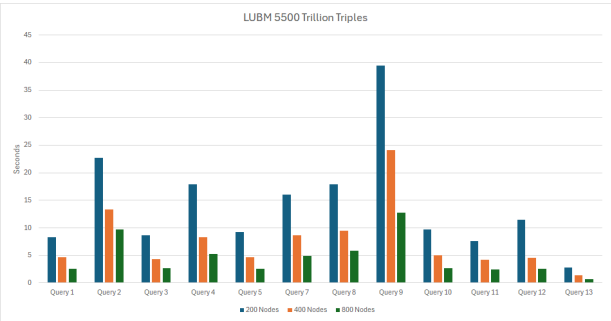


Fig. 2. LUBM5500 Trillion Triples Queries

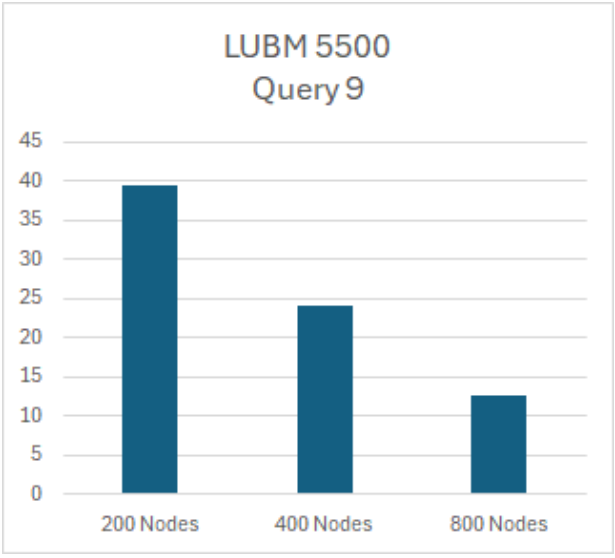


Fig. 3. LUBM5500 Query 9 Scaling

5.2 Image and Video Analysis

The second use case focused on an image and video analysis workflow that utilizes multiple AI models, both for search object detection and meta-data generation during the data load process as well as for target and feature extraction during queries. For the use case here, our queries focused on identifying people based upon specific characteristics, including predicted age, gender and facial sentiment. The intent of this workflow is to mimic a security analysis use case where users may need to analyze large amounts of image/video data to search for potential evidence. For example, looking for images where there may be an older male, a scared child and a vehicle.

The input dataset includes ∼90,000 images retrieved from the Google OpenImages datasets [15] using the open-images python package. The input image data load was performed using pretrained torchvision models [16] for object detection. The face/age/gender feature identification was done using torch models [16, 17]. Finally, the sentiment analysis was done leveraging the DeepFace python package [18].

Using AI for this use case highlights the ability of IDS to enable users to combine multiple AI powered UDFs into a single, complex workflow to analyze data at scale. Further, as will be shown in section 5.2.1, the query leverages both statically and dynamically loaded Python modules, illustrating the ease for users to add new Python UDFs to an IDS instance to utilize in a search.

5.2.1 Query Performance

For testing the performance of IDS for the image analysis use case, two different queries were used and each were executed with different combinations of the AI query optimizations from 4. The first query focused on finding images with two males, one between ages '25-32' and the other between '4-6'. The second query, shown in 1, is a simple extension of the first query with the addition of sentiment analysis to find the same individuals but only if they are both 'happy'.

Both queries use multiple pretrained AI models to perform inferencing on the given image as part of the FILTER expression evaluation. In fact, each UDF within the expression requires multiple AI models to be used – one for face detection and another for predicting the age/gender/sentiment. This means that for each solution being evaluated, the first query could perform up to 8 AI inferences and the second query, which includes the added sentiment analysis, could perform up to 12 AI inferences. For the 90,000 image dataset used in this test, there are 1,019,233 solutions that match the query and are passed to the FILTER operation. This means, at a minimum, there are ∼2 million AI inferences to perform, and in the worst case scenario, there would be ∼12 million AI inferences to perform in order to complete the FILTER. The ability to execute millions of AI inferences in parallel highlights the possibilities enabled by the performance and scalability of IDS. Since IDS will load balance the solution evaluations across images, the millions of AI inferences can be performed quickly, allowing for interactive analysis of large datasets even with multiple AI models being utilized.

For each query, three combinations of the AI query planning optimizations were tested. First, executing the queries with neither of the optimizations enabled. Second, enabling the reordering of FILTER expressions based upon the estimated UDF costs. Finally, enabling both the reordering of FILTER expressions and the rebalancing of intermediate solutions to evaluate using the estimated UDF costs.

As shown in Figure 4, for the query looking for only the two people of the given age and gender, both optimizations provided a performance improvement over the query without any optimizations. The expression reordering alone reduced query time from 121 to 96 seconds on 32 nodes, an ∼20% reduction in query time. Combining the reordering and rebalancing optimizations further reduced the query time to 73 seconds, providing an ∼40% reduction in query time. Even though these two models performed quite similar to each other, averaging between 0.08 - 0.1 seconds each, there was sufficient variance over the entire query for the optimizations to have a significant impact on query time.

For the query that included sentiment analysis, the results shown in Figure 4 again shows that both optimizations provided a performance improvement, though the difference between the gains from each was closer. In this query, the key improvement was in the reordering of expressions. The query in 1 shows how a user may write the query, processing all features for one individual first before processing the next. However, the profiling information tracked by IDS found that for most images, the sentiment analysis took the least time, followed by age prediction and then gender. This reordering reduced the query time on 32 nodes from 236 seconds to 125 seconds, an ∼47% reduction in query time and on 128 nodes the time was reduced from 108 seconds to 60 seconds, an ∼45% reduction in query time.

The performance improvements for these two AI powered queries, even while scaling up node counts, demonstrates the significant impact the optimizations IDS performs automatically for UDFs can have on reducing query times. These improvements become even more critical as users test new AI models and add models to their queries, making it nearly impossible for a human to track the performance of each to best optimize the queries.

```
select distinct ?img ?src ?box1 ?box2
where {
  ?img a <urn://image> ;
    <urn://source> ?src ;
    <urn://contains> ?obj1 ;
    <urn://contains> ?obj2 .
  ?obj1 a "person" ;
    <urn://bounding-box> ?box1 .
  ?obj2 a "person" ;
    <urn://bounding-box> ?box2 .
  filter(arq:user_func('predict_gender', ?src,
          '/container/models_age_gender_face', ?box1) = 'Male'
        && arq:user_func('predict_age', ?src,
            '/container/models_age_gender_face', ?box1) = '(25-32)'
        && arq:py_user_func('sentiment_udfs', 'get_sentiment', ?src,
            ?box1) = 'happy'
        && arq:user_func('predict_gender', ?src,
            '/container/models_age_gender_face', ?box2) = 'Male'
        && arq:user_func('predict_age', ?src,
            '/container/models_age_gender_face', ?box2) = '(4-6)'
        && arq:py_user_func('sentiment_udfs', 'get_sentiment', ?src,
            ?box2) = 'happy')
}
```

Listing 1. Query for two people of specific age/gender and both happy

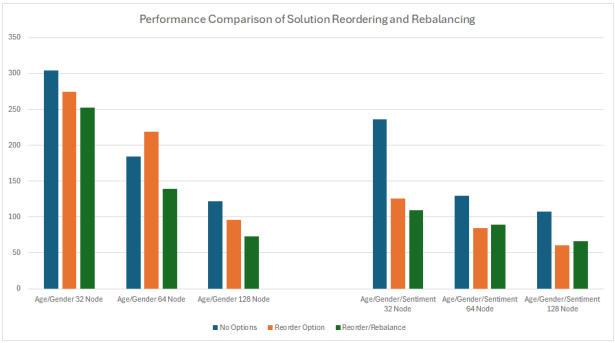


Fig. 4. Image Analysis with Rebalancing and Reordering Options

5.3 Life Sciences

The next workflow considered in our study relates to ongoing efforts with the National Center for Natural Products Research [19], where researchers are performing multiple drug-repurposing studies focused on leveraging natural products. For this study, we focused on the research related to the Adenosine A2A receptor (Uniprot identifier P29274 [20]) and finding alternatives to caffeine. The NCNPR workflow builds upon the drug discovery efforts that leveraged CGE during the Covid-19 pandemic to aid researchers in finding potential existing drugs that could be reused against Covid-19 [4]. The Covid-19 efforts used CGE to host a life sciences knowledge graph created from multiple public sources, which created a graph including over 100 billion medical facts. Table 1 shows the public datasets combined to create the life sciences knowledge graph used in the NCNPR use case. Further, the NCNPR use case adds structural information for the protein of interest to the graph from the Protein Data Bank (PDB) [21]. The size of the PDB data is considerably smaller since it is added for proteins as needed so it is not listed in table 1.

The NCNPR research expands the drug discovery workflow by utilizing the new capabilities of IDS for embedding AI models and HPC style simulations. These improvements to IDS, combined with the scalable performance, enable the researchers to perform the drug repurposing workflow at a larger scale beyond their current capabilities. For example, the NCNPR research currently runs on small clusters or even single node servers for doing simulations on a small

TABLE 1
Knowledge Graph Dataset Characteristics.

Dataset	Raw Size (disk)	Size (triples)	Source
UniProt	12.7 TB	87.6 Billion	[22]
ChEMBL-RDF	81 GB	539 Million	[23]
Bio2RDF	2.4 TB	11.5 Billion	[24]
OrthoDB	275 GB	2.2 Billion	[25]
Biomodels	5.2 GB	28 Million	[26]
Biosamples	112.8 GB	1.1 Billion	[27]
Reactome	3.2 GB	19 Million	[28]

number of compounds, often manually chosen by the researcher based upon individual knowledge from previous studies. By leveraging IDS, the drug discovery workflow could be expanded by multiple orders of magnitude in both the processing scale as well as the data being considered. The NCNPR drug repurposing workflow highlights each of the different search methods enabled by IDS and can be summarized by the following steps:

1) Search for all proteins related to adenosine a2a receptor, P29274 (”What-is”)
2) Find all proteins with a certain structural similarity to P29274 (”What-else”)
3) Find all compounds that target proteins similar to P29274 (”What-else”)
4) Join similar proteins with inhibiting compounds and use AI to predict drug-target binding affinity (DTBA) between compounds and P29274 (”What-if”)
5) Perform docking simulation between compounds and P29274 (”What-could-be”)

The steps for finding all proteins related to P29274, those with a structural similarity to P29274 and the compounds that target those proteins are nearly identical to what was done in our previous research for Covid-19 [4]. The primary difference is in the protein of interest. The query to find similar proteins utilizes the same Smith-Waterman (SW) algorithm [29] for comparing two protein sequences as was done for Covid-19 [4]. The new additions to the workflow focus on adding DTBA and docking simulations to the query to better refine search results. The DTBA UDF is a TensorFlow based function that uses a pretrained AI model to predict the binding affinity for a given protein and compound, utilizing the protein amino-acid sequence information and the compound’s 1-dimensional information (SMILES) [30]. The docking simulation is done by leveraging the Autodock Vina package to compute docking energy when performing blind docking with a given compound and the PDB for the protein of interest [31, 32]. In this case, blind docking refers to enabling the docking simulation to consider the entire 3-dimensional space of the protein structure when searching for docking locations, rather than narrowing the search to a specific range of x/y/z coordinates. This is crucial as part of the workflow to enable researchers to consider new compounds that may have different binding behaviors, but comes at a cost of computational expense.

The query in listing 2 shows the steps involved in the drug repurposing search, which is divided into an inner query and outer query. The inner query executes the following steps:

- Search knowledge graph of all reviewed proteins
- Search knowledge graph for all compounds with desired properties that inhibit found proteins
- Filter results by sequence similarity using SW, pIC50 for compound potency, and DTBA for binding-affinity prediction score

Once the inner query completes, the outer query executes docking on the unique compounds found with the P29274 protein of interest. The query is split into the inner and outer query to ensure the inner query removes any duplicate compounds to use for the docking simulation with the P29274 protein and remove as many potential compounds before performing docking.

For this drug repurposing query, there are four different UDFs involved, to perform the protein similarity using the Smith-Waterman algorithm, compute the pIC50 value for the compounds, predict the drug-binding target affinity (DTBA) and finally, perform the docking simulation. The UDFs are ordered the way they are intentionally based on their performance and likeliness of removing potential solutions. For the query shown in 2, there are ∼66 million proteins from Uniprot that must be compared to the P29274 protein using the SW UDF, which represents the largest number of UDF calls in the entire query. The time for each SW UDF is quite small, averaging less than 1 millisecond per calculation. The pIC50 cost is extremely small per compound (less than $1x10^{-5}$ seconds) and the number of unique compounds passed to DTBA is only in the tens of compounds since we FILTER results by the similarity between the protein targeted by a compound and the protein of interest (P29274). The DTBA prediction is orders of magnitude slower than SW, taking tenths of seconds per prediction, however, the number of DTBA executions is quite small after filtering by similarity. Finally, the docking simulation is the most expensive of each operation, taking multiple seconds each, but the number of compounds used in docking simulations is very limited after filtering by similarity and DTBA. The docking cost and impact on query performance is discussed in more detail below in 5.3.1.

```
select distinct ?label ?min_energy ?smiles ?pdbUrl
where {
  {
  select distinct ?label ?smiles ?pdbUrl
  where {
    # Get the sequence information for our P29274 protein of interest
    uni2:P29274 a core:Protein ;
        core:mnemonic ?mnem ;
        core:sequence ?isoform .
    ?isoform rdf:value ?seq .

    # Look up activities that target proteins. We do this here to make
    # sure we only select proteins that are targets for doing our comparison
    ?targetcmpt cco:targetCmptXref ?protB .
    ?target cco:hasTargetComponent ?targetcmpt .
    ?assay cco:hasTarget ?target .

    # Look up known proteins and get sequence values
    ?protB a core:Protein ;
        core:sequence ?isoformB .
    ?isoformB rdf:value ?seqB .
    ?rcsb <urn://uniprotId> ?mnem ;
       <urn://rcsbPDBUrl> ?pdbUrl .

    # Get our approved compounds with inhibition activities
    # on target proteins
    { ?activity cco:type 'Inhibition' } union { ?activity cco:type 'IC50' }
    ?activity a cco:Activity ;
        cco:hasMolecule ?molecule ;
        cco:value ?val ;
        cco:units ?unit ;
        cco:hasAssay ?assay .
    ?molecule rdf:type cco:SmallMolecule ;
        cco:highestDevelopmentPhase ?phase ;
        skos:prefLabel ?label ;
```

```
        resc:SIO_000008 ?attr .
    ?attr rdf:type resc:CHEMINF_000018 ;
        resc:SIO_000300 ?smiles .
    ?assay cco:assayType 'Binding' .

    # Filter the results based upon the SW similarity, the compound phase
    # and pIC50, and finally the DTBA predicted value.
    filter(arq:user_func('ssw', ?seq, ?seqB) >= 0.35
           && ?phase >= 3
           && (arq:user_func('pIC50', ?val, ?unit) >= 4.0
               || arq:user_func('pIC50', ?val, ?unit) = 0.0)
           && arq:user_func('dtba', ?smiles, ?seq) >= 6.0)
  # End the where clause for inner query
  }
  # End the inner query scope
  }

  # Use vina to compute binding energy. This uses the dynamic UDF code
  # path and calls the UDF in a module named crickett_vina_udfs.py.
  bind(arq:py_user_func('crickett_vina_udfs', 'compute_min_docking_energy_blind',
       'P29274', ?smiles, ?pdbUrl,
       '/lus/scratch/crickett/ncnpr/docking_results', ?label, 9, 3, 1, 0, true) as
       ?min_energy)
}
order by asc(?min_energy)
```

Listing 2. Query for drug repurposing with DTBA and Autodock Vina

5.3.1 *Query Performance*

To test the performance of the docking query in listing 2, we executed the query on the full life sciences knowledge graph at multiple node counts. Our evaluation of the docking query workflow focused on both the overall query performance time as well as the time spent for the FILTER operation since these two components comprise the significant majority of the query time. The query was executed on 64, 128 and 256 nodes using 32 images per node for a total of 2048, 4096 and 8192 total images, respectively.

Figure 5 shows the scaling performance of the drug repurposing query that includes DTBA and docking simulation UDFs. As the figure shows, the query does not appear to scale very well; however, further analysis of the query performance showed that this scaling limitation comes from the docking simulations. The inner query returns a given number of potential compounds to consider for docking, and the number of results is quite small relative to the available images executing IDS. For the query in listing 2, there are 55 compounds returned by the inner query that are used in docking simulations with P29274. Given our experiments use 2048, 4096 and 8192 total images, there is clearly very little work for most images in the outer query so most images wait for essentially 55 images, each doing one docking simulation for a given compound, to finish the outer query work. The time for each docking simulation can be many seconds and vary considerably – for the compounds in this query, the times would range from 31-44 seconds per compound. Given that the query times take 86, 72 and 62 seconds on 64, 128 and 256 nodes, respectively, the docking simulation cost of 31-43 seconds becomes the dominant factor. In this case, with so many idle images, it is not a matter of needing to better balance the work. In fact, it is very likely that the thousands of idle images could each do other parallel docking simulations if the set of potential candidate compounds was expanded while not significantly impacting the overall query time. While the total query time is bound by the slowest docking simulation (∼43 seconds), the rest of the query execution is approximately 43, 29 and 19 seconds for 64, 128 and 256 nodes, respectively, which shows a reasonable scaling.

The docking query time prior to the docking simulation is encompassed by the inner query and is dominated by the time to perform the FILTER operation that includes the three UDFs for SW, pIC50 and DTBA. Figure 6 shows the time for the FILTER operation at 64, 128 and 256 nodes, and demonstrates how well the FILTER scales with execution times of approximately 27, 18.5 and 7.7 seconds, respectively. Note that there is still variance in the run times due to the DTBA AI powered prediction, with most predictions taking approximately 1 second each but some taking possibly multiple seconds. The scaling numbers shown here for the FILTER operation highlights the ability for IDS to rebalance the operations based on the throughput of each image for the UDFs involved.

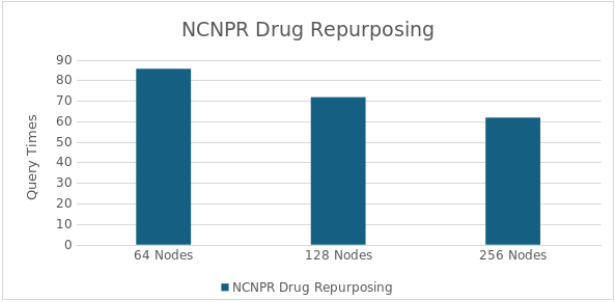


Fig. 5. NCNPR Drug Repurposing Query Times

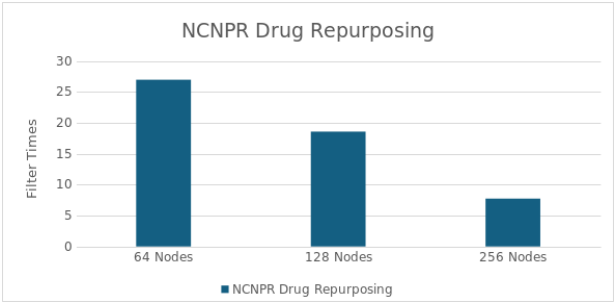


Fig. 6. NCNPR Drug Repurposing Filter Times

6 Conclusions

In this paper, we presented the HPE Intelligent Data Search (IDS) platform, which supports complex query workflows consisting of traditional graph searches, pretrained AI models and HPC style simulations. We described the architecture of IDS and the design choices made to enable key features, including: interaction with IDS from Python; ability to execute user programs and AI models during data ingest; and easily add new AI models and codes to power search workflows.

We also discussed query planning optimizations implemented in IDS to efficiently execute such complex workflows, in particular, at scale both in terms of system node count and number of AI inferences executing in parallel. We described how IDS can collect profiling information dynamically about user defined functions and utilize these statistics to optimize expression evaluations both through reordering expressions as well as by rebalancing solutions.

Finally, we demonstrated the performance of IDS with multiple use cases, including one that is a typical knowledge graph use case and two others that expand the graph query

with AI-powered, domain-specific and HPC style simulation user defined functions. We were able to show the ability for IDS to scale well, both in terms of total process count as well as number of AI inferences being performed, and quantify the impact of the query planning optimizations on these complex search workflows.

7 Future Work

As discussed in section 4.2, the solution rebalancing optimization within IDS should be able to efficiently execute queries even on heterogeneous systems. However, the benchmarks performed in 5.2.1 and 5.3.1 were only performed on a homogeneous Cray EX supercomputer. In the future, we intend to execute the workflows on a heterogeneous system to better illustrate the capabilities of IDS to effectively load balance in order to reduce query times.

Finally, our life sciences use case that included a docking simulation found that the performance and scaling was limited by the time of the slowest simulation, which is not surprising. We plan to investigate ways in which we could reduce the impact of the simulation, possibly through caching and reuse of results from prior simulations, as a way to reduce the impact on total query time and improve scalability.

References

[1] K. Maschhoff, R. Vesse, and J. Maltby, "Porting the Urika-GD graph analytic database to the XC30/40 platform," in *Cray User Group Conference (CUG '15)*, Chicago, IL, 2015.

[2] C. Rickett, K. Maschhoff, and S. Sukumar, "Optimizing the cray graph engine for performant analytics on cluster, superdome flex, shasta systems and cloud deployment," in *Cray User Group Conference (CUG '21)*, 2021.

[3] "Rpyc - transparent, symmetric distributed computing." [Online]. Available: https://rpyc.readthedocs.io/en/latest/

[4] C. Rickett, K. Maschhoff, and S. Sukumar, "Massively parallel processing database for sequence and graph data structures applied torapid-response drug repurposing," in *IEEE BigData 20202*, 2020.

[5] "Hugging face." [Online]. Available: https://huggingface.co

[6] "Apptainer." [Online]. Available: https://apptainer.org

[7] "Open MPI: Open Source High Performance Computing," 2021. [Online]. Available: https://www.open-mpi.org

[8] "libfabric." [Online]. Available: https://github.com/ofiwg/libfabric

[9] "shs-libcxi." [Online]. Available: https://github.com/HewlettPackard/shs-libcxi

[10] Y. Guo, Z. Pan, and J. Heflin, "Lubm: A benchmark for owl knowledge base systems," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 3, no. 2, pp. 158–182, 2005.

[11] R. Vesse, "Parallelized version of the Lehigh University Benchmark (LUBM) data generator," 2018, https://github.com/rvesse/lubm-uba.

[12] P. Castagna, C. Dollin, and A. Seaborne, "Vivisecting LUBM," HP Laboratories, Tech. Rep., Nov. 2009.

[13] "Trillion-triples benchmarking," Cambridge Semantics, Tech. Rep., Dec. 2016.

[14] C. Rickett, U. Haus, J. Maltby, and K. Maschhoff, "Loading and Querying a Trillion RDF Triples with Cray Graph Engine on the Cray XC," in *Cray User Group Conference (CUG '18)*, Stockholm, Sweden, 2018.

[15] I. Krasin, T. Duerig, N. Alldrin, V. Ferrari, S. Abu-El-Haija, A. Kuznetsova, H. Rom, J. Uijlings, S. Popov, S. Kamali, M. Malloci, J. Pont-Tuset, A. Veit, S. Belongie, V. Gomes, A. Gupta, C. Sun, G. Chechik, D. Cai, Z. Feng, D. Narayanan, and K. Murphy, "Openimages: A public dataset for large-scale multi-label and multi-class image classification." 2017. [Online]. Available: https://storage.googleapis.com/openimages/web/index.html

[16] G. Levi and T. Hassner, "Emotion recognition in the wild via convolutional neural networks and mapped binary patterns," in *2015 ACM on international conference on multimodal interaction*, 2015.

[17] "Torchvision: Pytorch's computer vision library," 2016. [Online]. Available: https://github.com/pytorch/vision

[18] S. Serengil and A. Ozpinar, "A benchmark of facial recognition pipelines and co-usability performances of modules," *Journal of Information Technologies*, vol. 17, no. 2, pp. 95–107, 2024. [Online]. Available: https://dergipark.org.tr/en/pub/gazibtd/issue/84331/1399077

[19] "National center for natural products research." [Online]. Available: https://pharm.olemiss.edu/ncnpr/

[20] U. Consortium, "Uniprot: the universal protein knowledgebase," *Nucleic Acids Research*, vol. 45, no. D1, p. D158, 2017. [Online]. Available: http://dx.doi.org/10.1093/nar/gkw1099

[21] "Rcsb protein data bank." [Online]. Available: https://www.rcsb.org

[22] U. Consortium, "Uniprot: the universal protein knowledgebase," *Nucleic Acids Research*, vol. 47, p. D506, 2019.

[23] D. Mendez, A. Gaulton, A. P. Bento, J. Chambers, M. De Veij, E. Félix, M. Magariños, J. Mosquera, P. Mutowo, M. Nowotka, M. Gordillo-Marañón, F. Hunter, L. Junco, G. Mugumbate, M. Rodriguez-Lopez, F. Atkinson, N. Bosc, C. Radoux, A. Segura-Cabrera, A. Hersey, and A. Leach, "ChEMBL: towards direct deposition of bioassay data," *Nucleic Acids Research*, vol. 47, no. D1, pp. D930–D940, 11 2018. [Online]. Available: https://doi.org/10.1093/nar/gky1075

[24] F. Belleau, M.-A. Nolin, N. Tourigny, P. Rigault, and J. Morissette, "Bio2RDF: Towards a mashup to build bioinformatics knowledge systems," *Journal of Biomedical Informatics*, vol. 41, pp. 706–716, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1532046408000415

[25] E. M. Zdobnov, F. Tegenfeldt, D. Kuznetsov, R. M. Waterhouse, F. A. Simão, P. Ioannidis, M. Seppey, A. Loetscher, and E. V. Kriventseva, "OrthoDB v10: sampling the diversity of animal, plant, fungal, protist, bacterial and viral genomes for evolutionary and functional annotations of orthologs," *Nucleic Acids Research*, vol. 47, no. D1, pp. D807–D811, 2019.

[26] V. Chelliah, N. Juty, I. Ajmera, R. Ali, M. Dumousseau, M. Glont, M. Hucka, G. Jalowicki, S. Keating, V. Knight-Schrijver, A. Lloret-Villas, K. Nath Natarajan, J.-B. Pettit, N. Rodriguez, M. Schubert, S. M. Wimalaratne, Y. Zhao, H. Hermjakob, N. Le Novère, and C. Laibe, "BioModels: ten-year anniversary." *Nucl. Acids Res.*, vol. 43, pp. D542–D548, 2015.

[27] S. Jupp, J. Malone, J. Bolleman, M. Brandizi, M. Davies, L. Garcia, A. Gaulton, S. Gehant, C. Laibe, N. Redaschi, S. M. Wimalaratne, M. Martin, N. Le Novère, H. Parkinson, E. Birney, and A. M. Jenkinson, "The ebi rdf platform: linked open data for the life sciences," *Bioinformatics*, vol. 30, no. 9, pp. 1338–1339, 05 2014. [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3998127/

[28] A. Fabregat, S. Jupe, L. Matthews, K. Sidiropoulos, M. Gillespie, P. Garapati, R. Haw, B. Jassal, F. Korninger, B. May, M. Milacic, C. D. Roca, K. Rothfels, C. Sevilla, V. Shamovsky, S. Shorser, T. Varusai, G. Viteri, J. Weiser, G. Wu, L. Stein, H. Hermjakob, and P. D'Eustachio, "The reactome pathway knowledgebase." *Nucleic Acids Res*, vol. 46, no. D1, pp. D649–D655, Jan 2018.

[29] M. Zhao, W.-P. Lee, E. P. Garrison, and G. T. Marth, "Ssw library: An simd smith-waterman c/c++ library for use in genomic applications," *PLoS One*, vol. 8, no. 12, 2013.

[30] H. Öztürk, A. Özgür, and E. Ozkirimli, "Deepdta: deep drug–target binding affinity prediction," *Bioinformatics*, vol. 34, no. 17, pp. i821–i829, 2018.

[31] A. F. T. J. Eberhardt, D. Santos-Martins and S. Forli, "Autodock vina 1.2.0: New docking methods, expanded force field, and python bindings," *Journal of Chemical Information and Modeling*, 2021.

[32] A. J. O. O. Trott, "Autodock vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization and multithreading," *Journal of Computational Chemistry 31*, 2010.