

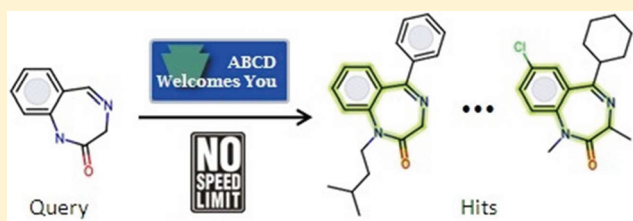
Efficient Substructure Searching of Large Chemical Libraries: The ABCD Chemical Cartridge

Dimitris K. Agrafiotis,^{*,†} Victor S. Lobanov,[†] Maxim Shemanarev,^{†,§} Dmitrii N. Rassokhin,[†] Sergei Izrailev,^{†,¶} Edward P. Jaeger,[†] Simson Alex,[†] and Michael Farnum[†]

[†]Johnson & Johnson Pharmaceutical Research & Development, L.L.C., Welsh & McKean Roads, Spring House, Pennsylvania 19477, United States

[§] Supporting Information

ABSTRACT: Efficient substructure searching is a key requirement for any chemical information management system. In this paper, we describe the substructure search capabilities of ABCD, an integrated drug discovery informatics platform developed at Johnson & Johnson Pharmaceutical Research & Development, L.L.C. The solution consists of several algorithmic components: 1) a pattern mapping algorithm for solving the subgraph isomorphism problem, 2) an indexing scheme that enables very fast substructure searches on large structure files, 3) the incorporation of that indexing scheme into an Oracle cartridge to enable querying large relational databases through SQL, and 4) a cost estimation scheme that allows the Oracle cost-based optimizer to generate a good execution plan when a substructure search is combined with additional constraints in a single SQL query. The algorithm was tested on a public database comprising nearly 1 million molecules using 4,629 substructure queries, the vast majority of which were submitted by discovery scientists over the last 2.5 years of user acceptance testing of ABCD. 80.7% of these queries were completed in less than a second and 96.8% in less than ten seconds on a single CPU, while on eight processing cores these numbers increased to 93.2% and 99.7%, respectively. The slower queries involved extremely generic patterns that returned the entire database as screening hits and required extensive atom-by-atom verification.



INTRODUCTION

Eric Schmidt, founder and executive chairman of Google, was once quoted as saying “Search is the ultimate expression of the power of the individual — using a computer, looking at the world, and finding exactly what they want.”¹ Indeed, it is hard to imagine today’s world without electronic search, and this is certainly no less the case for the chemical and pharmaceutical sciences. Chemical search, however, is much more challenging than ordinary text search because the objects of interest — molecules — are far more complex than words.

There are three main types of chemical search — exact structure, substructure, and similarity. Among them, exact search is the most straightforward, since all that is required is a method for generating a canonical representation of a molecule that is independent of the numbering of atoms and bonds. ‘Straightforward’ is, of course, a relative term, since there are a number of complicating factors, such as aromaticity and tautomerism, which introduce significant ambiguity and algorithmic complexity. But assuming consistent treatment, these canonical representations can be encoded as strings using SMILES² or an equivalent notation scheme and searched using conventional text-based techniques. Substructure and similarity searching are more challenging, in that the former involves a computationally formidable problem, and the latter is notoriously subjective.

The naïve approach to substructure search involves a linear scan through the database, where each record in the structure table is deserialized from an internal representation (such as SMILES, SD,³ or others) and matched against the pattern of interest. In graph theory, this is known as the subgraph isomorphism problem: given two graphs *M* and *Q*, the task is to find whether *M* contains a subgraph that is isomorphic to *Q*.^{4,5} Although the problem is known to be NP-complete, in the case of molecular graphs which are characterized by low connectivity, it is possible to develop algorithms with good average time performance.⁶ A number of such algorithms have been described, ranging from simple back-tracking⁷ and its variants,^{8–10} to more elaborate partitioning and relaxation schemes like those proposed by Sussenguth,¹¹ Figueras,¹² Ullmann,¹³ and Von Scholley.¹⁴

However, for chemical databases containing a large number of molecules, this linear approach is very time-consuming and inefficient. In most chemical database management systems (DBMS), the actual search involves two distinct steps: 1) screening and 2) verification. The screening phase involves a rapid search through the database to identify molecules that can be safely excluded from further consideration because they do not contain features that are present in the query pattern.¹⁵ For example, if the query

Received: September 3, 2011

Published: October 31, 2011

contains a chloro-phenyl group, any molecules in the database which do not contain aromatic or chlorine atoms can be easily identified and eliminated from further consideration. The molecules that pass this screening stage are then verified one-by-one using any of the aforementioned subgraph isomorphism algorithms. This molecule-by-molecule comparison is by far the rate limiting step in the entire process, so it is essential that the screening selectivity (defined as the fraction of true hits among the potential hits identified during screening) is as high as possible.

Screening is enabled through a process called indexing. In database vernacular, indices are data structures derived from the objects to be searched that improve the speed of data retrieval operations on a database table. Many chemical structure indexing algorithms are based on the generation of fixed-length vectors of 1 or 0 bits (key masks) designating the presence or absence of a particular structural feature (key) in the target molecule. When a database is being searched for a particular substructure pattern, the key mask of the pattern is first generated by the same algorithm that was used to generate the key masks of the molecules in the database and is then matched against them. Only if all the bits that are turned on (set to 1) in the pattern's key mask are also turned on in the molecule's key mask can the pattern be a substructure of the molecule. Screening based on structural key matching is very fast, although the screening selectivity depends greatly on the search pattern and the choice of the keys, which have to be carefully selected to maximize the selectivity of "typical" substructure queries against a given structure collection. Keys which are not sufficiently discriminatory (e.g., the presence of a carbon atom) will yield poor selectivity. Keys are typically selected based on their statistical occurrence in large chemical databases. The more advanced screening approaches utilize hierarchical tree structures, such as atom-centered fragments and ring descriptors. This class of approaches has been widely adopted in various systems.^{16–26}

Logical extensions of substructure keys are the so-called binary fingerprints.^{27,28} In fingerprints, bits at certain positions do not correspond to certain structural features explicitly defined by a domain expert. Instead, fingerprinting algorithms traverse all possible paths up to a certain length in the molecule and compute hash values from certain properties of the atoms and bonds encountered in each visited path. A primitive example of such a hashing function would be the sum of all atomic numbers of the non-hydrogen atoms in the path from length 0 (atoms) to a predefined maximum length N . After hash values are computed for each traversed path, they are mapped onto specific positions in a bit vector of a certain predefined length, and the corresponding bits are set to 1. Such a mapping is often carried out with a pseudorandom number generator using the hash value as the seed. A variant of the fingerprint generation algorithm, instead of traversing the paths, computes hash values from properties of each atom combined with the properties of its neighbors and properties of the bonds between them up to a certain level.

Fingerprints possess the same important feature structural keys do: a given search pattern can be a substructure of a molecule only if all bits set to 1 in the pattern's fingerprint are also set to 1 in the molecule's fingerprint, which means that fingerprints can be used for screening the same way structural keys are. Each method has its advantages and disadvantages. Structural keys tend to be shorter (usually a few hundred features/bits), but they are subjective and lack generality. Conversely, the features derived from exhaustive subgraph enumeration tend to be a lot

more numerous but are unbiased and do not require prior knowledge.

Regardless of whether structure keys or fingerprints are used for substructure screening, on modern computer hardware with gigabytes of RAM the keys or fingerprints encoded as bitsets of moderate length (usually, ranging from 512 to 2,048 bits per molecule) can fit in RAM even for a large database containing millions of compounds. This greatly simplifies screening, which can be performed entirely in memory by using simple bitset operations. However, folding a fingerprint into a bitset of a manageable size invariably results in loss of information, which leads to a decrease in screening selectivity.²⁹ To address this limitation, Baldi and co-workers have developed lossless fingerprint compression algorithms by combining the statistical properties of fingerprints with integer entropy encoding, resulting in storage efficiency that is comparable to or even better than the widely used fixed-length 1,024 bit fingerprints.³⁰

A very different approach that was specifically designed for relational databases was recently reported by Golovin and Henrick.³¹ This method casts the subgraph isomorphism problem into a standard SQL query without necessitating external function calls. By analyzing the symmetry of a query molecule and by performing breadth-first searches, their implementation on relational DBMSs allows a typical search of 1 million compounds to be performed in a few seconds.

In this paper, we describe the substructure search capabilities of ABCD,^{32,33} an integrated drug discovery informatics platform developed at Johnson & Johnson Pharmaceutical Research & Development, L.L.C. ABCD uses the same two-stage approach for substructure searching that is utilized by most other chemical DBMSs (screening and verification) but introduces several algorithmic enhancements to improve the efficiency of the search. In the remaining sections, we provide a detailed description of the key components of our solution, including the indexing strategy for efficient file-based screening, the pattern mapping algorithm for solving the subgraph isomorphism problem during verification, the encapsulation into an Oracle cartridge, and the cost estimation scheme used by the Oracle cost-based optimizer. In addition, we demonstrate how this functionality can be used for both client and server-based searching and delivered consistently to our entire scientific community through a layer of elegant graphical user interfaces.

METHODS

1. Key Generation. Our approach to screening is based on the premise that, if a certain feature (implicitly identified by its hash value) occurs in a search pattern N times, only those molecules in which the same feature occurs N or more times can match that pattern.^{34,35} For example, suppose we are given the following search pattern: $N-N-C$, and we use the simple hashing function we have mentioned as an example before, that is, the sum of atomic numbers of all non-hydrogen atoms in a path from length 0 to 2: $\{7,2\}, \{6,1\}, \{14,1\}, \{13,1\}, \{20,1\}$. Apparently, the screening would eliminate from consideration such molecules as NO ($\{7,1\}, \{8,1\}, \{15,1\}$), since in the collection of hash values computed for this molecule the value of 7 is encountered only once, and the values of 6, 13, 14, and 20 are not present at all. However, such a molecule as $N=N-C$ will pass the screening and will have to be eliminated in the next step using atom-by-atom subgraph matching.

Our indexing algorithm resembles binary fingerprints in that it relies on exhaustive enumeration of various types of features and subgraphs and uses hashing to convert these features into long (8-byte) integer codes. The algorithm computes hash codes and counts for five types of structural features: atoms, bonds, rings, paths, and clusters. Each key is a 64-bit integer where the highest 6 bits reserved for the key type (atom = 1, bond = 2, ring = 3, path = 4, cluster = 5; the extra bits are reserved for future types) and the rest of the bits are set using the procedures described below.

Atoms. Atoms are enumerated, and a set of 32-bit hash values is computed for each one. The following atomic properties are taken into account by the hashing algorithm: atomic number, charge, total attached hydrogen count, aromaticity, "aliphaticity", and isotope mass. A separate hash value, a predefined group number, is added to the set of hash values computed for an atom if that atom belongs to one of the predefined atom groups commonly searched for (for example, [F,Cl,Br,I], [Na,K], etc.). The 32-bit atom format is "...mm mmmm mmmh hhcc ccAa nnnn nnnn", where m is isotope mass, h is H count, c is charge, A is aliphatic flag, a is aromatic flag, and n is atomic number. Keys are computed for several common combinations of atomic properties in order to increase screening selectivity with substructure queries of varying degree of specificity, for example (using the SMARTS notation) [#6] for any carbon, c for aromatic carbon, [cH] for an aromatic carbon with exactly one hydrogen atom attached to it, etc. Each 32-bit hash code computed for an atom is combined with the atom key type constant (see above) into a 64-bit key by using the binary *shift* and *or* operations.

Bonds. Bonds are enumerated and keys are computed for each bond in the molecule as follows. Two sets of atomic hash values, one for each of the two atoms connected by the bond, are computed as described above for atoms. Then each possible pair of hash values for atom 1 and atom 2 is combined into a 32-bit integer by using a crc32 algorithm.³⁶ Each such hash value is combined with the bond key type constant into a 64-bit integer by using the binary *shift* and *or* operations. Additional keys are computed by combining each hash value obtained as described above with an integer representing the bond order, again by using the *shift* and *or* operators. This makes it possible to efficiently screen for "any bond between two particular atoms" as well as "a bond of specified order between two particular atoms".

Rings. All rings up to size 8 are enumerated, and for each ring keys are computed by combining the ring size with the crc32 of the vector of group numbers to which each atom in the ring belongs (see Atoms section above), the ring aromaticity flag, and a vector of bond orders for all bonds in the ring. Again, separate hash values are computed for several common combinations of these values, which allows for efficient screening of rings specified with various degrees of specificity (e.g., "any ring of size 4", "aromatic ring of size 6", "aliphatic cyclopentane ring", etc.). Each such hash value is combined with the ring key type constant into a 64-bit integer by using the binary *shift* and *or* operations.

Paths. Path keys are computed by traversing each possible path up to length 4 in the molecular graph. A hash value is computed for each path by applying the crc32 function to the array of atomic numbers of all atoms encountered in the path. Since each path can be traversed in two different directions, hash value canonicalization is necessary so that the path keys are independent of atom and bond numbering. We canonicalize the hash values by computing the crc32 for the original and reversed vector of atomic numbers and choosing the smaller of the two.

The resulting hash code is combined with the path key type and the path length using binary *shift* and *or* operations to produce the 64-bit path key.

Clusters. Clusters refer to subgraphs that contain atoms with 3 or more bonds emanating from them. Cluster keys are computed by assembling a vector of atomic numbers, sorting it, and applying the crc32 function to the resulting list. Additional more general cluster keys are computed in a similar manner by using atom group numbers instead of the atomic numbers similarly to how it is described above for the atom keys.

2. Indexing. Typical indexing/screening algorithms store the fingerprints of the molecules in the database as binary sets (bitsets) and compare them one-by-one to the fingerprint of the query to determine whether they match. Although the comparison is carried out using very fast binary operations, the process can be inefficient for large databases, particularly if the fingerprints do not all fit in the computer's main memory.

We address this problem by using inverted bitsets, that is, by maintaining a binary mask for every unique key/count combination, where each bit represents a particular molecule in the database. Our algorithm proceeds as follows. Every molecule in the database is analyzed, and the set of keys described in the previous section along with the number of occurrences in that molecule are computed. The information for each key is stored as a triplet $\{h, c, m\}$, where h is the hash code, c is the count (number of occurrences), and m is the index of the molecule in the database (numbered sequentially from 1 to N). Once all the molecules have been processed, the triplets are sorted in ascending order, first by hash code, then by count. The sorted list is then traversed, and for each unique hash code, h , a series of binary masks, $M(h, cmin)$, are defined, where $M(h, cmin)$ contains the indices of the molecules for which the hash code h occurs at least $cmin$ times. Finally, the masks are saved on a pair of files, one (.iset) containing the actual masks, and the other (.iidx) containing the byte offset (pointer) of each mask in the .iset file. The need for the offset file comes from the fact that the masks do not necessarily occupy the same amount of space when written to disk but are stored intelligently so as to minimize disk space. If N is the total number of molecules in the database and K is the number of molecules that belong to the set represented by the mask, then the indices are saved in one of three ways: if $K < (N/32)$, we store the indices of the molecules that belong to the set as an array of integers; if $(N - K) < (N/32)$, we store the indices of the molecules that do not belong to the set (i.e., the set's complement, as that requires less space than storing the set itself), also as an array of integers; otherwise we store the set as a mask of 1's and 0's (e.g., 10010101...) in binary format.

Our implementation of the above algorithm is actually more involved. The triplets are stored in a vector of finite capacity depending on the amount of available memory, and when the capacity of that vector is about to be exceeded (say, after molecule i), the triplets are sorted and saved on a segment in a pair of temporary files, one containing the keys (.tiset) and one the file offsets (.tiidx). This vector is then cleared and the process resumes with the next molecule $i+1$. This process continues until all the molecules have been processed. Finally, the segments in the temporary files are merged to produce the aforementioned .iset and .iidx files. In addition, we have implemented a multi-threaded version, where each thread processes a subset of molecules and accumulates the keys (triplets) into its own local buffer. When all the molecules have been processed, the threads synchronize, the keys are merged, and the generation of the index

files proceeds in the manner describe above. The rate-limiting step in the entire process is the parsing/perception of molecules and the calculation of keys. A further optimization involves the use of binary-encoded molecules instead of SMILES or SD specifications, where all the necessary molecular perception flags (valence, rings, aromaticity, topological symmetry, stereochemistry) have been precomputed and stored in a compressed binary stream, which can be rapidly deserialized into a fully fledged Molecule object used for the key generation.

3. Screening. Given a database indexed in the manner described above, screening becomes a straightforward task. The process involves three initialization steps. First, the query pattern is analyzed and a set of hash code/count pairs $\{h(q), c(q)\}$ are computed exactly the same way as for the molecules in the database. Second, a binary set H that will contain the screening hits is initialized to length N (database size) and every bit is set to 0. Third, the database index files (*.iset* and *.iidx*) are opened, and all the keys are read and stored in a sorted list L as triplets $\{h, cmin, p\}$, where h is the hash code, $cmin$ is the minimum count, and p is the byte offset (pointer) of that record in the *.iset* file. Because the keys are already sorted in the index files and are read in sequence, the list L is already sorted by construction. Then, for each key $\{h(q), c(q)\}$ in the query, a search is performed to find a matching database key in L (this search is extremely fast since L is a sorted collection). If a matching key is not found, the search exits returning zero hits. If a matching key is found, the corresponding mask M is read from the *.iset* file by following the pointer p (direct file access through an *fseek* call), and H is replaced by the intersection of H and M (i.e., $H = H \& M$, where “&” is the binary “and” operation). If the updated H does not contain any “1” bits, the search exits returning zero hits. Otherwise, the process is repeated for all the remaining keys in the query. Upon return, the set H contains the screening hits which are then passed to the final stage for verification. The process outlined above is extremely fast (see timings in Results and Discussion below) and does not require multithreading.

4. Verification. The most definitive way to determine whether the specified substructure is indeed present in the target molecule is to perform explicit atom-by-atom mapping. As stated in the Introduction, pattern mapping attempts to match all possible chemical graphs represented by the query pattern against the chemical graph of the target molecule. Our pattern matching algorithm is based essentially on Ullman's algorithm¹¹ and includes several optimization steps to ensure that all matching subgraphs in the target molecule can be identified quickly and exhaustively.

In ABCD, substructure queries are specified using the SMARTS language,³⁷ a flexible and efficient notation scheme that uses rules that are straightforward extensions of SMILES. As the first step, the substructure pattern specified as a SMARTS string is converted into a graph where each node and each edge is associated with a logical expression representing the required properties of the respective atom or bond. For example, the SMARTS definition of the atom “[!C;R]” is translated into an expression tree: (NOT aliphatic carbon) AND in ring. Both the graph and the atom/bond property expression trees are stored in an in-memory structure to facilitate fast matching against the candidate molecules.

In order to assert whether the target molecule contains the query substructure, our algorithm traverses the target graph according to the paths of the query graph and verifies that every atom and bond in the target matches the properties of the corresponding atom and bond in the query. The traversal is done

by following all nonintersecting paths of the query graph in a depth-first manner and stops when all such paths have been successfully mapped onto the target structure or when all attempts to map the paths have been exhausted.

The number of paths that need to be traversed increases substantially with the size of the two graphs. In order to reduce the search space and minimize the number of paths to trace, we employ several optimizations. First, we compute compatibility matrices between the query and target atoms and bonds. The atom compatibility matrix contains a row for each atom in the query pattern and a column for each atom in the target molecule. A value of 1 in the (i,j) position of the matrix signifies that the i th atom in the query is compatible with or can match the j th atom in the target. The atom compatibility matrix is populated by applying logical expressions associated with each query atom to each of the atoms in the target molecule. The bond compatibility matrix is computed in a similar way, but in addition to comparing the bond properties we also check that the atoms connected by the bond are also compatible. A row of zeros in the atom/bond compatibility matrix means that one of the atoms/bonds in the query cannot be matched to any of the atoms/bonds in the target and a negative outcome of the pattern matching can be deduced without traversing any paths. The compatibility matrices are also used during path traversal. When the i th query atom is matched with the j th target atom, we only need to consider bonds sprouting from the j th atom that are compatible with the next bond in the pattern in order to identify the next target atom to traverse.

The second optimization is related to choosing the starting point for the subgraph matching. For a typical organic molecule, if we start matching the graphs from a carbon atom, chances are there will be many matching candidates and hence many possible starting points and many paths to traverse before the mapping succeeds or fails. If we select the starting atom at random, the probability that we will pick a carbon is high since it is the most common element in organic molecules. Thus, in order to reduce the number of starting candidates, we use as starting point the query atom with the highest atomic number, as heavier atoms tend to be less frequent.

There could be multiple ways to map a substructure pattern onto a molecule. Finding all possible matches is a substantially longer process than finding the first one. However, in some applications we need to enumerate all possible matches, for example when mapping a reaction pattern. Our algorithm quickly returns the first match but keeps a detailed trace of the mapping process so that finding a subsequent match can resume from the previous one without having to retrace any already visited paths. The mapping context also helps to avoid path intersections and allows backtracking of the path traversal process when the next matched node leads to a dead end. In fact, the path tracing algorithm is implemented as a recursive function that makes depth-first path exploration and backtracking simple and “natural” form a programming point of view.

5. Oracle Cartridge Implementation. The indexing, screening, and verification algorithms described above lend themselves to a variety of implementations, including relational databases and flat files. For flat files, we have developed a command line utility called *dbsubsearch* which is available on both Windows and Linux platforms and whose syntax is described in the Supporting Information. Here, we focus our attention on the Oracle implementation.

Oracle provides interfaces that allow one to extend the functionality of the Oracle server to make use of domain knowledge

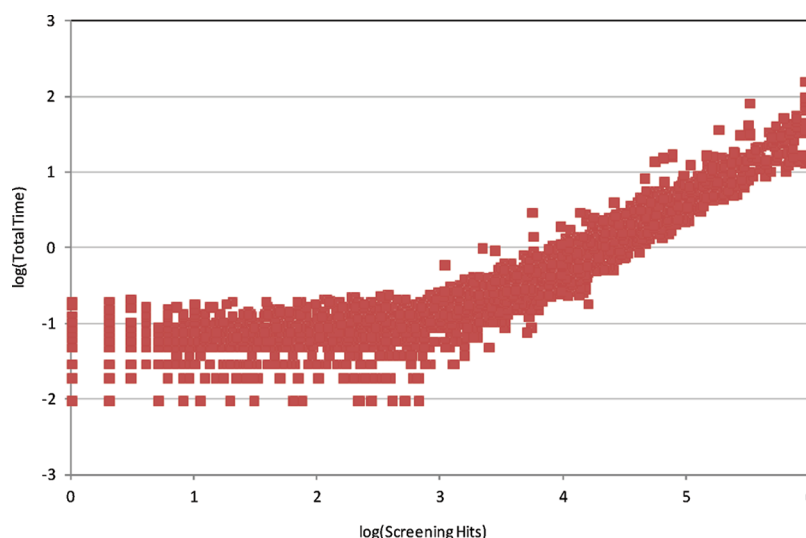


Figure 1. Log–log plot of total search time vs the number of screening hits. Below 1,000 hits, the time required to perform the search appears to be constant. Above 1,000 hits, it scales linearly with the number of screening hits.

accumulated and developed by the user. In this work we take advantage of three technologies provided by Oracle, namely, extensible execution environment, extensible indexing, and extensible optimizer. Extensible execution environment provides the means to make function calls to external procedures, implemented in our case in C++. Extensible indexing allows for custom built-in indices, called domain indices, which are used by Oracle much the same way as the regular indices such as b-trees. Finally, an extensible optimizer interface provides the means of feeding to Oracle information about the cost of searching and the selectivity of a predicate (in this case, a substructure pattern) for the given table. These extensions are then accessed using the regular SQL syntax.

6. Cost-Based Optimization. At the time when the Oracle cost-based query optimizer (CBO) is presented with a SQL query, it attempts to come up with an optimal execution plan to minimize the time required to execute the query. For example, if a query calls for selecting all rows from a table where the chemical structure column matches a given substructure and was tested in a given experiment, the CBO evaluates several possible execution plans such as (a) use the substructure search index to find all rows matching the substructure, then use the experiment id index to find all rows matching the experiment id, and finally perform an “and” operation on the row ids to obtain the final result; or (b) first find the row ids that match the experiment id and then perform exact substructure matching on chemical structures in those rows to arrive at the final result. In cases not involving an Oracle cartridge, the CBO performs optimization based on database statistics that must be collected on all relevant tables and is part of Oracle database administration machinery. However, in order to be able to compare the execution plans involving a substructure search to each other, the CBO needs three key pieces of information from the substructure search engine. The first piece is the cost of accessing the index, including the screening and verification stages. The second piece is the average cost of an exact substructure match, which in our case is also a part of the index verification cost. Finally, a critical piece of information is the selectivity of a substructure search, i.e., the fraction of rows in the table selected by a given substructure query. In the example above, if the selectivity of the substructure

search query is low (many rows are selected), and there are also many rows selected by the experiment id, scenario (a) above may be preferable. However, if the experiment id only returns a few rows, while the cost of accessing the substructure search index is high due to the verification stage, scenario (b) would be more favorable.

When statistics are collected on the substructure domain index, a set of a few thousand SMARTS patterns is used to sample the performance of the index. For each SMARTS in this set, several values are recorded, including the time of performing the substructure search using the index, the time of performing the screening stage only, the average time of performing the exact substructure match to a small random subset of structures in the indexed table, the number of rows in the table, the number of rows that matched the SMARTS, the number of rows that was retrieved by the screening stage, the number of rows with missing structures, etc. These values are further used for estimating the quantities required by the CBO.

Index Cost. Since the cost of accessing the substructure search index is usually dominated by the verification stage, it can be modeled as a function of the number of rows that pass the screening stage. The number of screened rows is estimated as a product of the number of rows in the table and the screening selectivity obtained as described below. A typical log–log plot of the time required to perform the substructure search using the domain index as a function of the number of screened rows is presented in Figure 1. Each point represents the timing of a query with a single SMARTS. In the chemical cartridge, the index cost is modeled as a constant A if the number of rows that passed the screening is below 1,000 and estimated from a linear model otherwise

$$\log(\text{index cost}) - \log(A) = K * (\log(\text{number of screened rows}) - \log(1000)) \quad (\text{Eq.1})$$

The values of A and K are stored with the index statistics and are used in the functions called by the CBO to compute the index cost for a specific query.

Cost of Exact Matching. The average time of an exact substructure match is estimated during statistics collection by

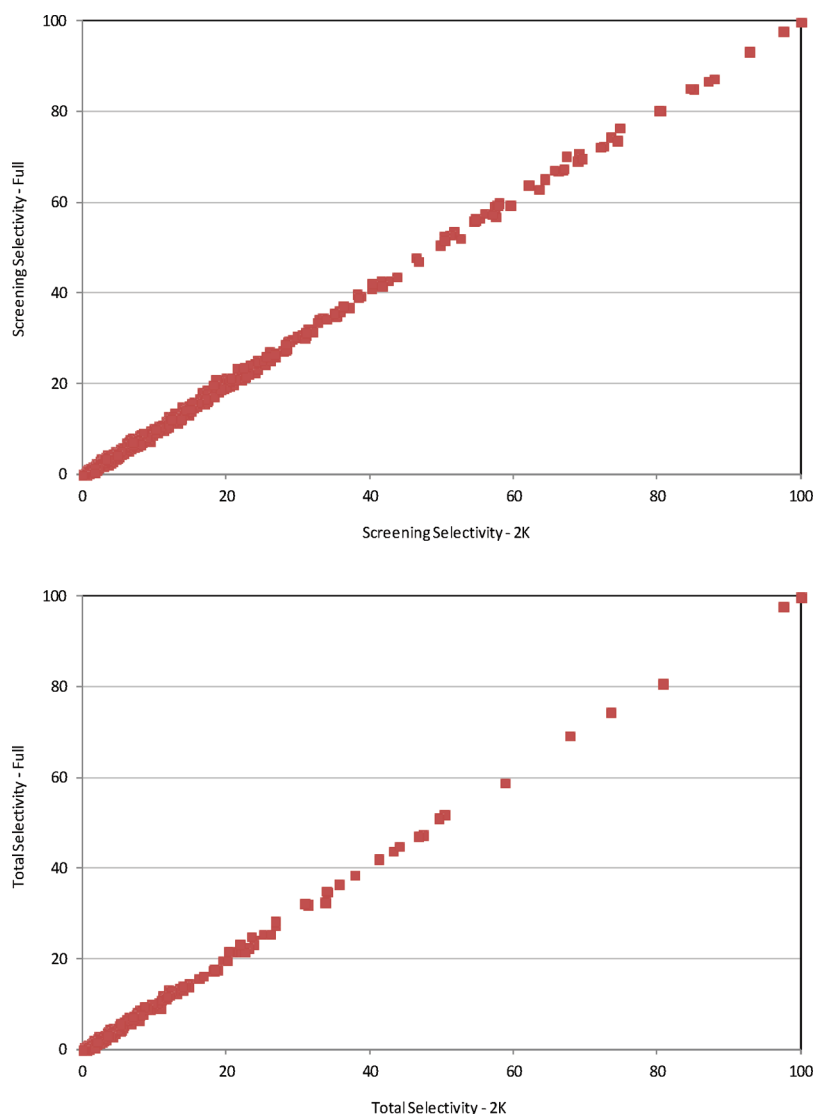


Figure 2. Screening selectivity (a) and total selectivity (b) of all the queries against the entire database of 907,558 molecules (Y axis) and a random sample of 2,000 molecules (X axis). The correlation is excellent in both cases, suggesting that a small random sample is sufficient to estimate the selectivity of the query.

measuring the time required to match several SMARTS patterns to each of approximately 2,000 randomly chosen structures from the indexed table. The resulting value is used by the CBO irrespective of the actual SMARTS in a given query. A more involved approach would be to precompute and cache the actual average matching time for a number of commonly used SMARTS. This option will be evaluated and implemented in a future version.

Selectivity. There are two types of selectivity that need to be estimated: the screening selectivity, i.e., the fraction of rows returned by the screening phase of the substructure index search, and the total selectivity, i.e., the fraction of rows matched by the substructure. The former is used in the evaluation of the verification cost, while the latter is used directly by the CBO. A good estimate of both selectivity values comes from direct calculation of the selectivity on a relatively small random subset of structures in the indexed table. The random subset is extracted and indexed on disk at the time of the table index creation. When a new SMARTS pattern appears in a query, the full substructure

search is first run on this subset, and the calculated selectivity is the estimate for the whole table. To take advantage of the fact that users generally use the same substructures in their queries, the computed values are also cached in an index-organized table for fast lookup by the SMARTS. This is especially efficient since the majority of SMARTS submitted in the queries are actually SMILES and can therefore be represented in the canonical form to avoid duplicates. Figure 2 shows the screening and total selectivity for all the queries against the entire database of 907,558 molecules versus a random sample of 2,000 structures (0.2%). The correlation is very high in both cases, suggesting that 2,000 randomly chosen molecules provide an excellent estimate of selectivity regardless of the structure of the query.

One concern with this approach is performance of the query, since every substructure search query needs to compute the exact selectivity on the subset, unless the SMARTS is already present in the cache table. However, the speed of the substructure search engine allowed for this approach to be viable. As demonstrated below, computing the selectivity of a typical query on 2,000

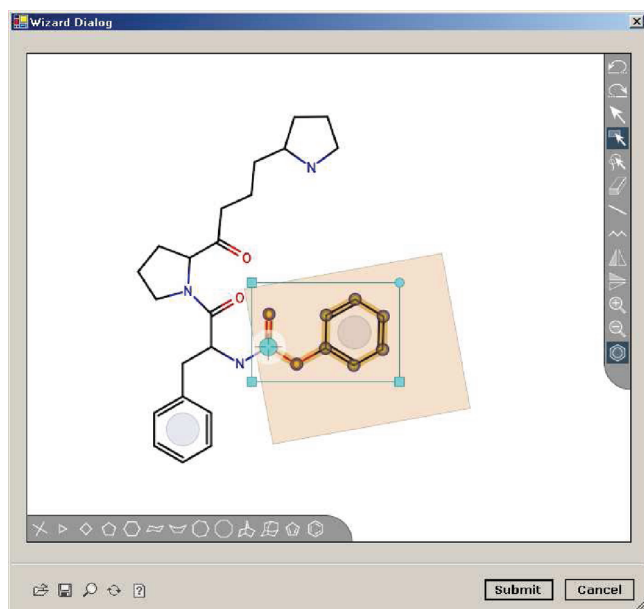


Figure 3. Screenshot of the ABCD/3DX chemical sketcher.

molecules takes less than 0.01 s, which is perfectly acceptable for the benefit of accurately estimating the total cost of accessing the domain index and the selectivity of the query.

It is worth noting that all the values above are used by the CBO in conjunction with other statistics to select an optimal query execution plan among a normally limited set of options. In many cases, choosing the best option available may not be as critical as not choosing a bad one. The cost and selectivity estimates are therefore required to be stable and fast, while a high accuracy of the estimate itself would only matter at the boundaries of cost landscapes of various execution plans.

7. ABCD Implementation. ABCD is a decision support system and was not designed for transactional processing. The master transactional database at Johnson & Johnson Pharmaceutical Research & Development, L.L.C. that determines uniqueness and subsequently persists the structures of all chemical compounds synthesized by discovery scientists is called the Global Compound Registration System (GCRS). Any client service that provides chemical structure information to the scientific community, such as ABCD, will have to maintain a local repository of structures that is a faithful copy of the GCRS data. ABCD uses a set of extraction, transformation, and loading (ETL) workflows to synchronize the chemistry record changes in GCRS with those in ABCD. The CTAB information provided by this feed from GCRS is converted into a SMILES representation by calling a structure conversion function provided in our chemistry cartridge. The loading of structures into ABCD involved an initial bulk load, followed by twice-daily incremental updates for newly registered or updated structures from GCRS.

The ABCD database table column with the SMILES representation of the structures has the Oracle domain index applied to it. This domain index is built on top of the file-based searchable index described above. Oracle interacts with this file via the Oracle Data Cartridge Interface (ODCI), which allows for the common actions on index related to definition and maintenance. When the domain index is created, all structures in the indexed column are converted from SMILES into a binary stream representation and stored in a structure file accessible by the cartridge.

The Oracle rowids and the sequential number of each molecule object in this file are recorded in an index-organized table for fast lookup by rowid. The rowids are also stored in a separate file in the same order as the molecules in the structure file and indexed for lookup by the molecule's sequential number in the structure file. The latter two structures allow the domain index to quickly locate a specific rowid in the file when it needs to be invalidated. Finally, the structure file is indexed for substructure searching as described above. In addition, a random subset of structures is similarly extracted and indexed for use during the estimation of query selectivity for the CBO. When the domain index is accessed with a SMARTS, a file-based substructure search is performed, and a list of sequential numbers of molecules that match the SMARTS is computed. This list is used to retrieve the corresponding rowids from the rowid file and then pass them back to Oracle as the result set.

Unlike a file-based collection, a database table is not static and is constantly changed by the users who may add new structures as well as delete or modify previously entered ones. The domain index has to reflect these changes in real time. In our implementation, newly inserted and updated structures are converted to the binary stream representation and stored in a separate index-organized table in binary format. Exact substructure matching is performed on all of these structures in addition to the process described above. Deleted and the original versions of the updated structures are marked as invalid in the file containing the rowids and are excluded from results.

The new structures are not prescreened, and to avoid the overhead, they have to be added to an indexed set from time to time. However, since the number of compounds in ABCD exceeds 3 million, rebuilding the full domain index is a long process that cannot be executed on a daily basis. This impracticality is compounded by a geographically widespread user base contributing to a very narrow time window in which such a rebuild can be executed. On the other hand, hundreds of new compounds are registered by Johnson & Johnson Pharmaceutical Research & Development, L.L.C. scientists every day that need to be exposed to search clauses coming in from various user interface driven applications such as the ABCD wizard discussed below. We have addressed this issue by maintaining two sets of index files that can be searched in tandem. The first set is the primary index file that is created as part of ODCI definition process for the current set of structures. The primary index data are rebuilt on a quarterly or per need basis. The second set contains the index data for structures that are inserted and updated on a daily basis. The modifications index data is rebuilt daily. The quarterly rebuild of the primary index results in the merging of the modifications index data into the primary index file, leaving an empty modifications file ready to store the index data for new or updated structures for the next quarter.

8. Query Interfaces: Third Dimension Explorer (3DX) and the ABCD Query Wizard. Our cartridge functionality is exposed to the end-users through various querying interfaces developed mostly as plug-ins to Third Dimension Explorer (3DX), a .NET application developed internally at Johnson & Johnson Pharmaceutical Research & Development, L.L.C. and designed to address a broad range of data analysis and visualization needs in drug discovery. 3DX is a table-oriented application, similar in concept to Microsoft Excel. Much of 3DX's analytical power comes from its ability to handle very large data sets through its embedded database technology, to associate custom cell renderers with each data type in the spreadsheet, and to visualize the

entire data set using a variety of generic and domain-specific viewers.^{38–43} The program offers a full gamut of navigation and selection options, augmented through linked visualizations and interactive filtering and querying.

3DX features a plug-in architecture that allows new functionality to be developed independently of the main application and delivered to the user either automatically or on a per-need basis. One of these plug-ins is a query wizard that allows scientists to interrogate the ABCD warehouse and return the results and associated metadata in a variety of tabular formats. The filtering

capabilities available in the interface are extensive and designed to take advantage of the breadth of information available in the warehouse. From the biology side, results can be filtered by protocol, target, result type, discovery stage, screening technology, team, and site where the assay was conducted as well as by the numeric values for experimental results and run dates for assays. From the chemistry side, filtering can be either directly related to the chemical nature of the molecule being tested, such as molecular weight, salt form, exact structure, or substructure, or involve organizational dimensions, such as the synthesizing

Table 1. Search/Screening Results for a Diverse Set of Queries in a Chemical Database Containing ~1 Million Molecules Using Various Methods^{a,b}

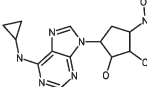
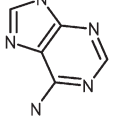
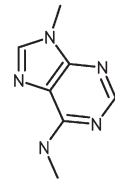
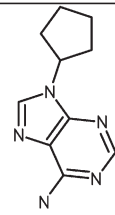
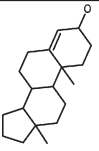
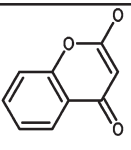
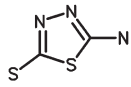


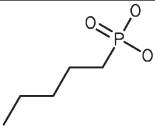


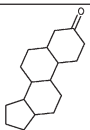
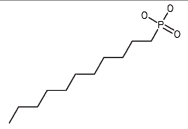

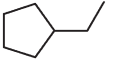
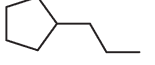
	Query	ABCD Screening Hits	ABCD Verification Hits	CACTVS Screening Hits	MACCS ¹ Screening Hits	FP2 Screening Hits
1		1	1	1	162	1
2		2739	2724	13045	21488 ²	2777
3		553	481	12785	17153	557 ²
4		108	84	430	11823 ²	178
5		5601	73	419	6987	691
6		2173	16	11261	15582	15 ²
7		1071	1071	5904	3785 ²	1071

Table 1. Continued

	Query	ABCD Screening Hits	ABCD Verification Hits	CACTVS Screening Hits	MACCS ¹ Screening Hits	FP2 Screening Hits
8		148	148	149	154	140 ²
9		2740	2740	8982	6221	3090
10		819	295	6854	6221	303
11		206	150	178	1566	545
12		28663	28662	28974 ²	76328 ²	37793 ²
13		7335	456	2872	160660	12133
14		179	71	2118	6221	159
15		29357	25297	40141 ²	442230	36260
16		27646	20365	38955 ²	442230	29016 ²
17		26038	16075	33219 ²	442230	29016

^a Superscript 1. In order to make the comparison valid, we had to exclude 36 MACCS fingerprint bits corresponding to the patterns with explicitly specified hydrogen atom counts: 28, 34, 43, 53, 54, 68, 69, 74, 82, 84, 86, 90, 91, 93, 100, 104, 108, 109, 111, 114, 115, 116, 118, 128, 129, 131, 132, 138, 139, 141, 147, 149, 151, 153, 155, 160. ^b Superscript 2. A small number of false negatives were observed (usually 1–2, 15 in the worse case of MACCS keys with pattern 2). This has been traced to a slightly different perception of aromaticity between different toolkits and does not bias the screening efficiency comparison.

chemist, creation date, internal projects associated with the compound, and internal or public aliases.

Depending on the complexity of the query, the options specified in the wizard are translated into one or more SQL statements executed in parallel against the data warehouse (the only exception is a BLAST⁴⁴ search where the chemical agent is a biologic, which is accomplished through a Web

service request.) This separation is done for two main reasons. The first is to keep the complexity of the SQL issued to the database low, so that query generation and debugging is simplified and indexing schemes on the database can be verified. The second is to improve performance both in executing the query on the server and in assembling sections of the results on the client.

Table 2. Search Results for the Golovin-Henrick (GH) and Hicks-Jochum (HJ) Queries

query	SMARTS	NKeys ^a	NScrHits ^b	NVerHits ^c	TScr1 ^d	TTot1 ^e	TTot8 ^f
GH1	ONC1CC(C(O)C1O)[n]2cnc3c(NC4CC4)ncnc23	115	1	1	0.16	0.18	0.16
GH2	Nc1ncnc2[n]cnc12	44	2739	2724	0.08	0.97	0.13
GH3	CNc1ncnc2[n](C)cnc12	57	553	481	0.09	0.14	0.11
GH4	Nc1ncnc2[n](cnc12)C3CCCC3	63	108	84	0.11	0.13	0.11
GH5	CC12CCC3C(CCC4=CC(O)CCC34C)C1CCC2	48	5601	73	0.06	1.45	0.25
GH6	OC2=CC(=O)c1c(cccc1)O2	67	2173	16	0.13	1.05	0.20
GH7	Nc1nnc(S)s1	52	1071	1071	0.06	0.61	0.08
GH8	C1C2SCCN2C1	73	148	148	0.11	0.13	0.11
GH9	CP(O)(O)=O	43	2740	2740	0.05	0.38	0.06
GH10	CCCCCP(O)(O)=O	57	819	295	0.06	0.11	0.07
GH11	N2CCC13CCCCC1C2Cc4c3cccc4	59	206	150	0.09	0.12	0.10
GH12	s1cncc1	38	28663	28662	0.06	1.53	0.33
GH13	C34CCCC1C(CCC2CC(=O)CCC12)C3CCCC4	42	7335	456	0.06	0.98	0.26
GH14	CCCCCCCCCCCCP(O)(O)=O	57	179	71	0.06	0.08	0.08
GH15	CC1CCCC1	19	29357	25297	0.05	1.63	0.38
GH16	CCC1CCCC1	19	27646	20365	0.05	2.16	0.49
GH17	CCCC1CCCC1	19	26038	16075	0.05	2.77	0.60
HJ1	[*D1]=[#6D2]-[#6D2]-[#6D2]=[#8D1]	8	715119	1	0.05	16.63	3.52
HJ2	[*D1]=[#6]-[#6]-[#6]=[#8D1]	8	715119	39220	0.03	21.27	4.38
HJ3	[#6]-[#6D2]-[#6D2]-[#6D2]-[#6D2]-[#6D2]-[#6D2]- [#6D2]-[#6D2]-[#6D2]-[#6D2]-[#6D3](=[#8D1])-[#8]	14	159673	540	0.05	10.87	2.19
HJ4	[#6]-[#6]-[#6]-[#6]-[#6]-[#6]-[#6]-[#6]-[#6]- [#6]-[#6]-[#6](=[#8D1])-[#8]	14	159673	3786	0.05	12.61	2.52
HJ5	[#6D1]-[#6]-1-2-[#6]-[#6D2]-[#6D2]-[#6D3]-1- [#6D3]-3-[#6D2]-[#6D2]-[#6D3]-4-[#6D2]-[#6]- [#6D2]-[#6D2]-[#6]-4-([#6D1])-[#6D3]-3-[#6D2]-[#6D2]-2	13	9373	868	0.02	1.13	0.25
HJ6	[#6]-[#6]-1-2-[#6]-[#6]-[#6]-[#6]-1-[#6]-3-[#6]-[#6]-4- [#6]-[#6]-[#6]-[#6]-4-([#6])-[#6]-3-[#6]-[#6]-2	13	9373	3067	0.00	1.99	0.43
HJ7	[#6D2]-1=[#6D2]-[#6D2]-2-[#6]-[#6]-[#6D3]-1-[#6D2]-2	11	9418	0	0.03	0.28	0.13
HJ8	[#6]-1=[#6]-[#6]-2-[#6]-[#6]-[#6]-1-[#6]-2	11	9418	2918	0.03	0.45	0.12
HJ9	[*;D1,D2,D3,D4]-[#6D3]1=;[#6D2][#6D2]=; [#6D2][#6D2]=;[#6D3]1-[*;D1,D2,D3,D4]	8	771856	216885	0.03	48.98	9.92
HJ10	[*;D1,D2,D3,D4]-[#6D3]1=;[#6D2][#6D2]=; [#6D2][#6D3](-[*;D1,D2,D3,D4])=;[#6D2]1	8	771856	77527	0.05	57.40	11.59
HJ11	[*;D1,D2,D3,D4]-[#6D3]1=;[#6D2][#6D2]=; [#6D3](-[*;D1,D2,D3,D4])[#6D2]=;[#6D2]1	8	771856	274811	0.05	53.60	10.77
HJ12	[*;D1,D2,D3,D4]-[#6D3]-1=[#6D3](-[*;D1,D2,D3,D4])- [#6D2]-[#6D2]=[#6D2]-1	8	47284	13	0.03	1.81	0.44
HJ13	[#6]-1=[#6]-[#6D2]-[#6]=[#6]-1	7	47343	52	0.03	1.61	0.41
HJ14	[#8]-[#6D3]-1-[#8D2]-[#6D3](-[#6D2]-[#8D2]-[#6D3]-2- [#6D3](-[#8D1])-[#6D3](-[#8D1])-[#6D3](-[#8D1])-[#6D3]-2- [#8D1])-[#6D3](-[#8D1])-[#6D3](-[#8D1])-[#6D3]-1- [#8D1])	28	155	0	0.05	0.10	0.06
HJ15	[#8]-[#6D3]-1-[#8D2]-[#6D3](-[#6D2]-[#8D2]-[#6D3]- 2-[#8D2]-[#6D3](-[#8])-[#6D3](-[#8])-[#6D3](-[#8])- [#6D3]-2-[#8])-[#6D3](-[#8])-[#6D3](-[#8])-[#6D3]-1- [#8]	28	155	0	0.05	0.10	0.05
HJ16	[#6D1]-[#7D3]-1-[#6D3]2=;[#6D2][#6D2]=;[#6D2][#6D2]=; [#6D3]2-[#7]-[#6D3]=;3[#6D2]=;[#6D2][#6D2]=;[#6D2][#6D3]-1=;3	23	7676	2	0.05	0.44	0.15
HJ17	[#6D1]-[#7D3]-1-[#6]2=;[#6][#6]=;[#6][#6]=;[#6]2-[#7]- [#6]=;3[#6]=;[#6][#6]=;[#6][#6]-1=;3	23	7676	5	0.05	0.44	0.14
HJ18	[#6D1]=[#6D2]-[#6D3]-1=[#6D3](-[#6D1])-[#6D3]=;2[#6D2]=; [#6D3]3[#7D2][#6D3](-;[#6D2][#6D3]- 4=;[#7D2][#6D3](-[#6]=[#6D3]-4-[#6D1])=; [#6D2][#6D3]5=;[#6][#6D3](-[#6D1])=;[#6D3]([#6D2]=; [#6D3]-1-1[#7D2]=;2)[#7D2]5)[#6D3](-[#6D1])=;[#6D3]3- [#6D2]=[#6D1]	21	432	10	0.03	0.06	0.03

Table 2. Continued

query	SMARTS	NKeys ^a	NScrHits ^b	NVerHits ^c	TScr1 ^d	TTot1 ^e	TTot8 ^f
HJ19	[#6]-1=[#6]-[#6D3]2=:[#6][#6D3]3=:[#6][#6]=:[#6D3]([#6D2]=:[#6D3]4-[#6]=[#6]-[#6D3]([#6]=:[#6D3]5[#6]=:[#6][#6D3]([#6]=[#6D3]-1=:[#7D2]2)[#7D2]5=:[#7D2]4)[#7D2]3	20	474	63	0.03	0.09	0.05
HJ20	[*;D1,D2,D3,D4]-[#6D3]=:1[#6D2]=:[#6D2][#8D2][#6D3]=:1-[*;D1,D2,D3,D4]	17	99015	660	0.05	3.63	0.83
HJ21	[#6]=:1[#6]=:[#6][#8D2][#6]=:1	16	99070	37426	0.05	3.57	0.83
HJ22	[*;D1,D2,D3,D4]-[#6D3]=:1[#6D2]=:[#6D2][#7D2][#6D3]=:1-[*;D1,D2,D3,D4]	17	162503	164	0.05	4.96	1.19
HJ23	[#6]=:1[#6]=:[#6][#7D2][#6]=:1	16	162550	14814	0.05	5.04	1.19
HJ24	[*;D1,D2,D3,D4]-[#6D3]=:1[#6D2]=:[#6D2][#16D2][#6D3]=:1-[*;D1,D2,D3,D4]	17	53617	2133	0.05	1.93	0.47
HJ25	[#6]=:1[#6]=:[#6][#16D2][#6]=:1	16	53663	46018	0.03	2.08	0.49
HJ26	[*;D1,D2,D3,D4]-[#6D3]-1=[#6D3](-[*;D1,D2,D3,D4])-[#34D2]-[#6D2]=[#6D2]-1	16	5	0	0.03	0.03	0.03
HJ27	[#6]-1=[#6]-[#34D2]-[#6]=[#6]-1	15	5	0	0.03	0.03	0.03
HJ28	[*;D1,D2,D3,D4]-[#6D3]=:1[#6D2]=:[#6D2][*D2][#6D3]=:1-[*;D1,D2,D3,D4]	6	433329	2969	0.03	13.53	2.81
HJ29	[#6]=:1[#6]=:[#6][*D2][#6]=:1	5	433582	97276	0.03	15.12	3.13

^a Number of keys. ^b Number of screening hits. ^c Number of verified hits. ^d Screening time (s) using the command line utility on a single thread on the IBM D20. ^e Total search time (s) using the command line utility on a single thread on the IBM D20. ^f Total search time (s) using the command line utility on 8 threads on the IBM D20.

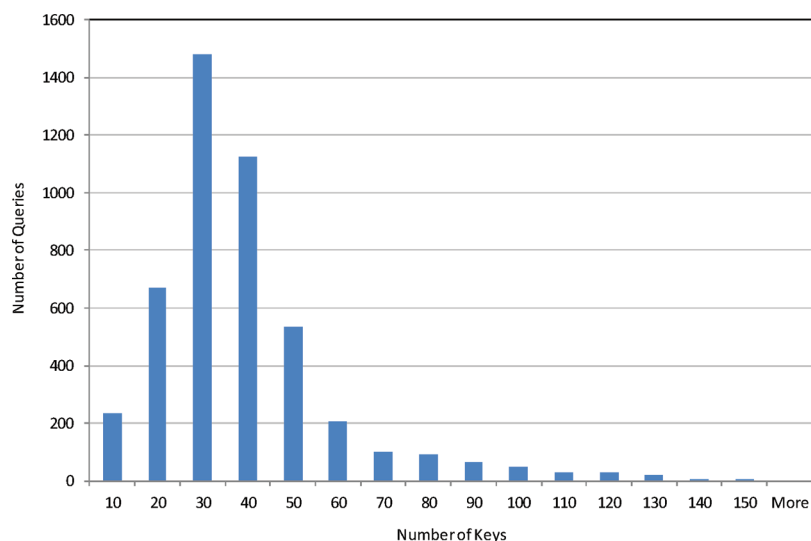


Figure 4. Histogram of the number of keys generated by all the query patterns.

9. Sketcher. Editing of substructures is enabled through a dedicated chemical sketcher, which is also embedded in the ABCD wizard (Figure 3). Our sketcher allows for interactive drawing and editing of valid molecules as well as query patterns. The sketcher component is written in C++ and is based on the chemical structure handling classes of the Mt library,⁴⁵ while the graphical rendering is implemented using Anti-Grain Geometry.⁴⁶ The sketcher component is written in a way that it can be included in a graphical Windows application implemented in either C++ or Microsoft .NET.

With the sketcher, users can readily compose substructure queries of arbitrary complexity by drawing a substructure and further

restricting the search pattern by specifying desired properties for certain atoms and bonds. Examples of such properties for the atoms are allowed/disallowed atom lists, aromatic character (membership in an aromatic ring), specific isotope mass, attached hydrogen count, charge, ring membership with an optional ring size qualifier, the number of total connections, and the number of attached heavy atoms. The editable bond properties include bond order, aromatic character, and ring membership. The sketcher translates visually edited queries into SMARTS strings for submission to the substructure search code.

To ensure compatibility with other popular desktop applications, our sketcher allows structures to be copied and pasted to

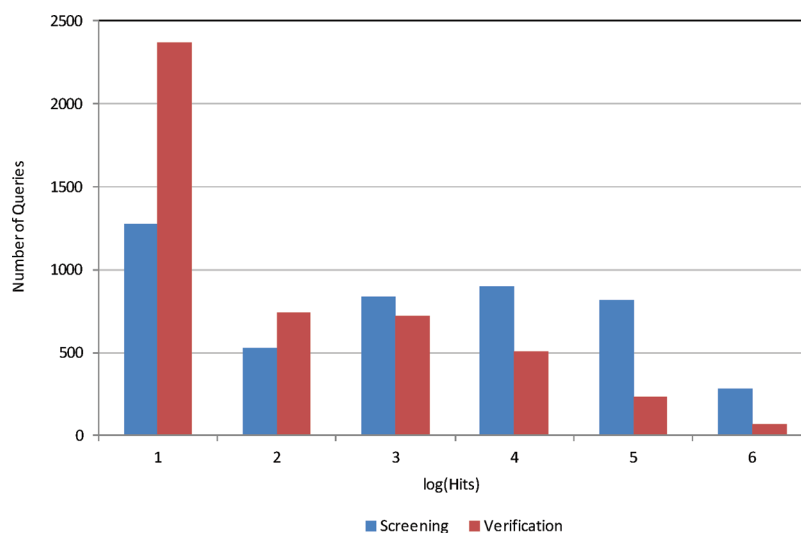


Figure 5. Histogram of the number of screening and verification hits on a log scale.

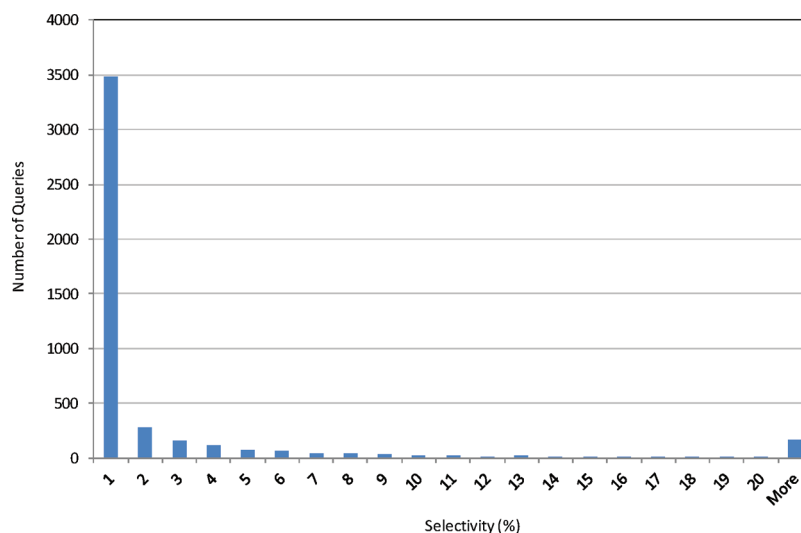


Figure 6. Histogram of the selectivities of our screening algorithm.

and from ISIS Draw⁴⁷ and ChemDraw⁴⁸ and also supports exporting the images as bitmaps or Windows metafiles so that they can be used in publications and presentations. A number of other rendering and customization options are also provided.

DISCUSSION

The performance of our substructure search algorithm was tested on a database of 907,558 unique molecules, derived from the collection used by Golovin and Henrick²⁹ after removing all stereochemically redundant SMILES and a few structures that could not be processed by our internal software. Golovin and Henrick's data set was originally compiled from the Open NCI database and a few other public sources and contained 931,007 molecules.

This database was interrogated with three different sets of queries. The first, abbreviated HJ, consisted of 29 queries used by Hicks and Jochum to compare the MACCS, DARC, HTSS, CAS, and S4 substructure search systems.²² These queries derived

from sixteen different structures, each presented in two different forms: the first had at most two free sites (atoms with variable number of substituents) and represented a "typical" query a chemist would pose to a database, whereas the second had many free sites and was designed to stress-test the performance of these algorithms.

The second set, abbreviated GH, consisted of 14 queries used by Golovin and Henrich to test their relational algorithm,²⁹ augmented by three additional queries with a specific topological signature that is relevant for a new generation of keys that we are currently developing. In these substructures, every atom with incomplete valence was assumed to be a free site that could match any number of substituents. Moreover, the aromaticity of the atoms in the supplied SMARTS was honored so that, for example, the pattern "CP(O)(O)=O" would not match phenylphosphonic acid.

The third set, abbreviated ABCD, consisted of 4,584 substructure search queries submitted to the test instance of the ABCD data warehouse from October 2008 until May 2011. Although we

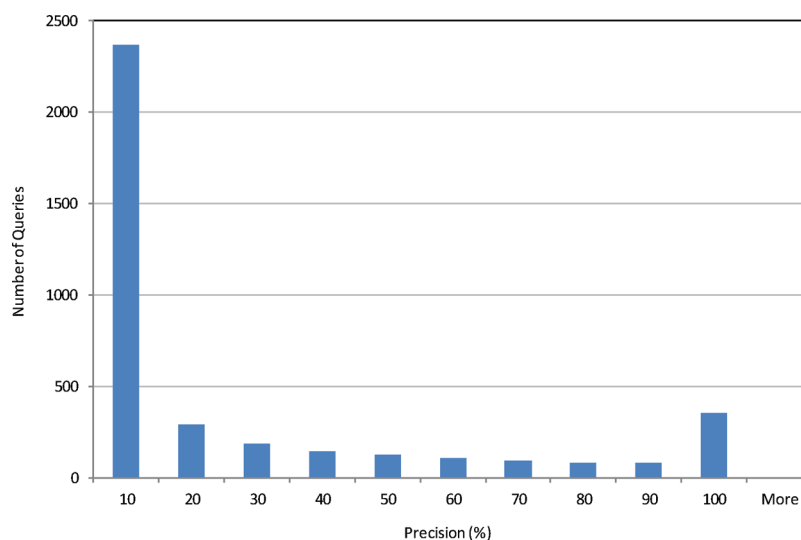


Figure 7. Histogram of the precisions of our screening algorithm.

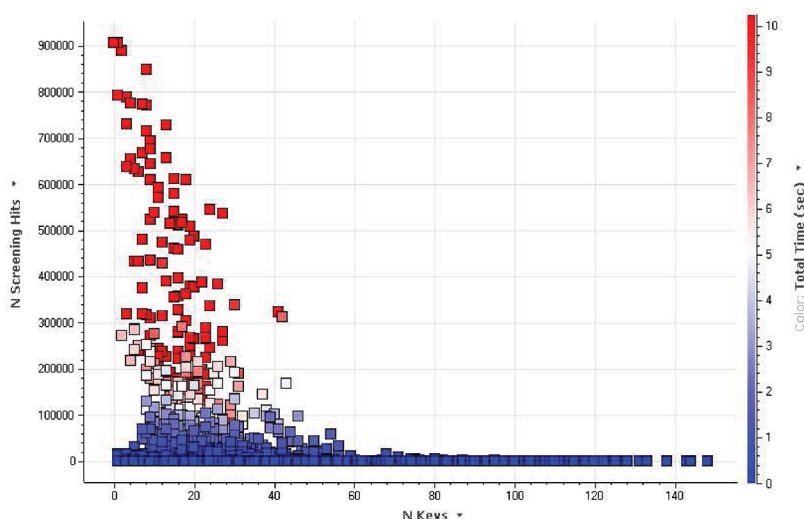


Figure 8. Relationship between number of screening keys (X axis), number of screening hits (Y axis), and total query time using the command line utility (color scale).

cannot disclose their chemical structures for obvious reasons, this data set is extremely valuable as it represents a very large and diverse set of real-life queries obtained during extensive user-acceptance tests of ABCD over a long period of time and spanning a broad spectrum of topologies, projects, locations, and use cases. We must point out that the ABCD queries differ considerably from the HJ and GH queries in the way the SMARTS were encoded. The SMARTS produced by our sketcher use the broadest possible definition of atoms and bonds allowed by the SMARTS standard. For example, a single carbon atom without an explicit aromatic flag is encoded as "[#6]" so as to match either an aliphatic or an aromatic carbon. Similarly, double bonds in aromatic rings (drawn in their Kekule forms) are encoded as "=", so as to match either double or aromatic bonds on the target. A benzene ring is thus encoded as "[#6]=,1[#6]=,[#6][#6]=,[#6][#6]=,1". This definition allows the pattern "[#6][#6]" to be matched against all six bonds of a phenyl group and "[#6]-[#6]" to be matched against only the

three bonds which are explicitly drawn as single. Because this definition is more permissive, the resulting queries are less selective and thus more expensive to execute. Although this problem can be easily mitigated by explicitly specifying the aliphatic/aromatic character of the atoms and bonds in question, the vast majority of the ABCD queries that contain aromatic rings use the generic descriptions (e.g., [#6], etc.)

The time required to index the database using the command line utility was 558 s, of which 539 s was consumed reading the molecules, constructing the keys, and writing the temporary index files (.tiset and .tiidx), and 19 s was consumed merging these temporary files, generating the inverted bitsets, and producing the final index files (.iset and .iidx). Indexing on the Oracle instance involved a small additional overhead associated with populating various tables and establishing the appropriate links. The entire database generated 44,202 distinct keys (hash values) and 127,217 unique key/count pairs. The sizes of the .iset and .iidx index files were 328 Mb and 2.9 Mb, respectively, whereas

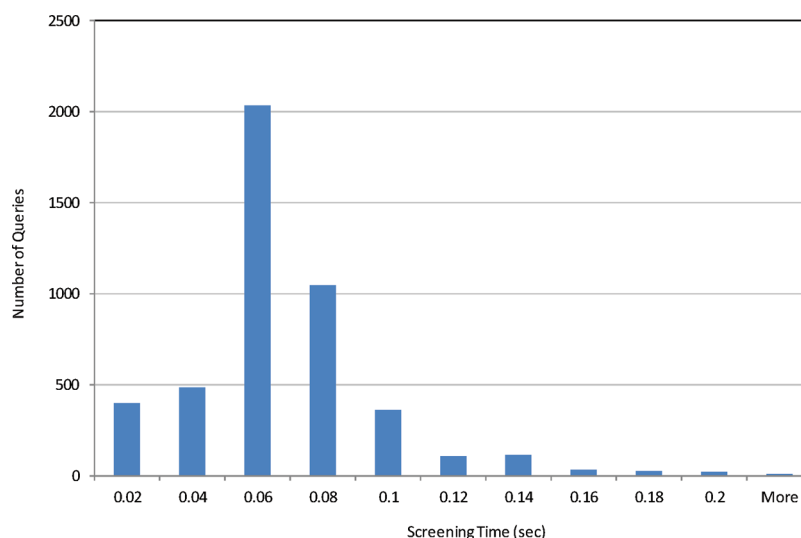


Figure 9. Histogram of screening times using the command line utility (in s).

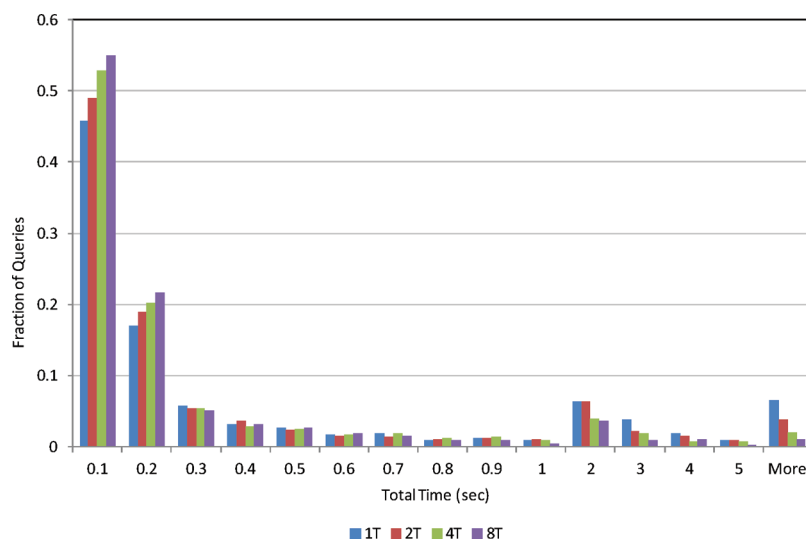


Figure 10. Histogram of total search times using the command line utility with 1, 2, 4, and 8 threads (in s).

the size of the original SMILES and binary-encoded molecule files were 39 Mb and 376 Mb.

Database search methods are typically evaluated using three key metrics: 1) *screening efficiency* or *selectivity*, defined as the ratio of screening hits over the total number of molecules in the database; 2) *recall*, defined as the ratio of verified hits over the number of true hits in the database; and 3) *precision*, defined as the number of verified hits over the number of screening hits. A good screening method is one that does not produce any false negatives and minimizes the number of false positives. The higher the number of false positives, the more time is spent on expensive atom-by-atom matching. In our case, recall was assessed by exact atom-by-atom mapping of each query against every molecule in the database and was confirmed to be 100% in all cases (i.e., screening with our keys did not produce any false negatives).

The screening efficiency of our keys was compared against three other structural keys commonly used for substructure searching,

namely, the 166-bit MACCS keys,^{49,50} the CACTVS keys used in PubChem and computed using Xemistry's CACTVS toolkit,⁵¹ and the 1024-bit path-based FP2 fingerprints as implemented in the OpenBabel toolkit.^{52,53} The comparison was carried out using only the queries in the GH set, since the other queries contained our permissive specification of atoms and bonds (e.g., [#6]) that could not be handled with the interfaces available for any of the other key generation methods. The searches were performed by binary matching of the fingerprints computed for the query molecules against the fingerprints computed for the database molecules. The results are summarized in Table 1.

For the majority of the queries, the selectivity of our keys was greater than the other three methods. CACTVS produced fewer screening hits for only three out of 17 queries. For two other queries the performance was comparable, and for the remaining 12, CACTVS generated 3 to 24 times more screening hits than our keys. The difference was even more pronounced with MACCS keys, though in this case the comparison is compromised by the

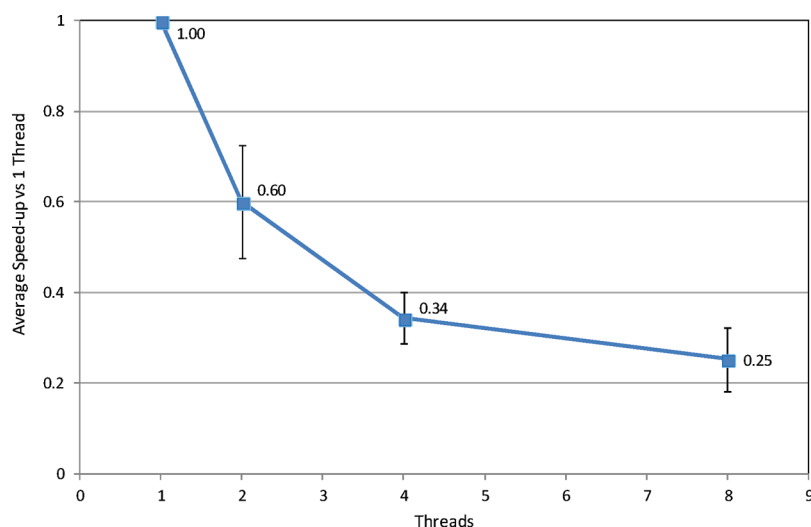


Figure 11. Multithreading speed-up for verification, defined as the ratio of the verification time using N threads over verification time using 1 thread.

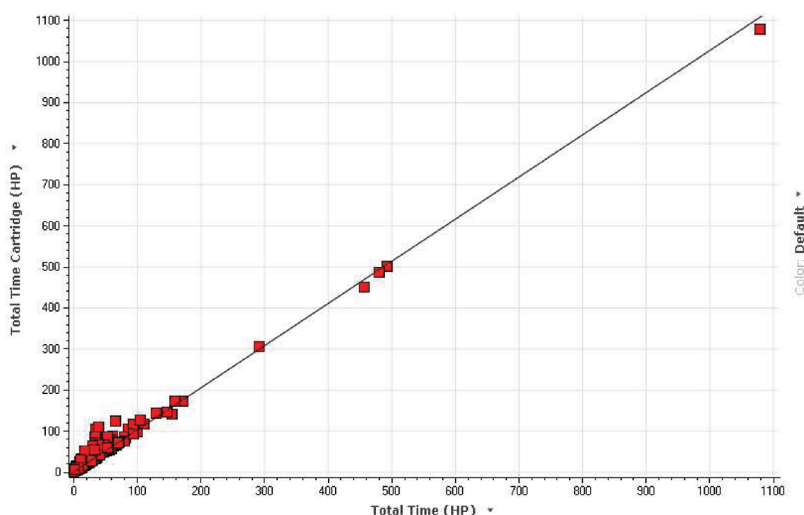


Figure 12. Total search time (s) using the command line (x -axis) and Oracle (y -axis) implementations.

fact that we had to exclude 36 MACCS keys which corresponded to the patterns with explicitly specified hydrogen atom counts (e.g., [CH₃]) that could not be used in the present context (bits 28, 34, 43, 53, 54, 68, 69, 74, 82, 84, 86, 90, 91, 93, 100, 104, 108, 109, 111, 114, 115, 116, 118, 128, 129, 131, 132, 138, 139, 141, 147, 149, 151, 153, 155, 160). FP2 selectivity was better than our keys for four of the queries, comparable for three, and worse for the remaining ten by 10%–132%. The hit lists produced by each of these screening methods were verified one-by-one using our own pattern matching algorithm, and in some cases a very small number of false negatives were discovered (1–2, and in one case 15). Although in theory the number of true hits should be the same for all three methods, some discrepancies arise due to slight differences in the perception of aromaticity. On the basis of these results, we conclude that the screening efficiency of our keys is very competitive.

For the remaining discussion, we turn our attention to the performance of our method on the three sets of queries described above using both the command line and Oracle implementations. The results are summarized in Tables 1 and 2 and Figures 4–12.

The two implementations differed only in the amount of time required to complete the search; all the other statistics (screening keys, screening hits, verified hits, etc.) were identical since we used the same code base.

The topological complexity of the query is typically reflected in the keys generated by it. The distribution of the number of keys produced by the combined set of queries is illustrated in Figure 4 and shows a maximum of 148 and a median of 30. Eighteen queries generated no keys at all, while 21 queries generated only a single key. Typical examples of queries with no keys (which all came from ABCD) are “any atom” (*), “positively charged nitrogen” ([N+,n+]), and “carbon or nickel” ([#6,#28]). The distribution of keys among the three sets was quite different, with GH generating 19–115 keys with a median of 52, HJ generating 5–28 keys with a median of 14, and ABCD generating 0–148 keys with a median of 30. The query that generated the most keys among the two published sets is GH1 (see Table 2), which has an elaborate topological structure with many heteroatoms.

The number of keys determines to a large extent the number of screening hits, the distribution of which is illustrated in Figure 5

on a log scale (the plot shows both screening and verified hits). At the screening stage, 792 queries (17.1%) resulted in no hits and 1,274 queries (27.5%) resulted in ≤ 10 hits, while on the other end of the spectrum 1,097 (23.7%) queries resulted in $>10,000$ hits, 282 (6.1%) resulted in $>100,000$ hits, and 18 queries returned the entire collection of 907,558 molecules. The number of true hits following verification was substantially lower, with 1,575 queries (34%) returning no hits, 2,368 queries (51.2%) returning ≤ 10 hits, 297 queries (6.4%) returning $>10,000$ hits, and 68 queries (1%) returning $>100,000$ hits. The median number of screening and verified hits was 447 and 9, respectively. The distributions of the selectivity and precision of our screening algorithm are illustrated in Figures 6 and 7, respectively. While in the large majority of cases our method is very selective and very precise, there are certain types of queries that return an inordinate number of hits and require very fast verification in order to complete in a reasonable amount of time. These results make it clear that chemical database management systems must handle gracefully not only the "average" query but also the worst case scenario.

The relationship between screening keys, screening hits, and overall screening time is illustrated in Figure 8. As is evident from this plot, queries containing a large number of screening keys (X axis) produce a small number of screening hits (Y axis) and are completed in a very short period of time (color). Conversely, queries that produce a very large number of hits tend to have relatively few keys and take substantially longer to execute.

As seen in Figure 9, the screening step was extremely fast regardless of the query. The median screening time was 0.06 s and the maximum 0.22 s. This part is largely i/o-bound, as the vast majority of time is spent retrieving the inverted bitsets from disk. Generally, the greater the number of keys, the more bitsets need to be retrieved, and the greater the screening time.

For queries returning more than a modest number of hits, the total query execution time is dominated by verification (atom-by-atom matching). As illustrated in Figure 10, 45.7% of the queries were completed in 0.1 s or less on a single thread (screening plus verification), 80.7% in 1 s or less, and 96.8% in 10 s or less. Fortunately, verification is almost perfectly parallelizable, so the use of multiple CPUs and/or multiple cores can help reduce the amount of time considerably. As illustrated in Figure 10, when the verification is split over 8 threads, 55.0% of the queries were completed in 0.1 s or less, 93.2% in 1 s or less, and 99.7% in 10 s or less. The average speed-up with multithreading (defined as the ratio of verification time with N threads over verification time with 1 thread) is 0.6 for 2 threads, 0.34 with 4 threads, and 0.25 with 8 threads (Figure 11). The most expensive query required 626 s on one CPU and 132 s on 8 CPUs (4 CPUs \times 2 cores). The time required for verification depends not only on the number of screening hits but also the complexity of the query pattern; on a single CPU, the median number of verifications per second was 16,708 and the maximum was 82,283.

The timings reported above were obtained with the command line utility on an IBM ThinkStation D20 equipped with a 2.40 GHz quad-core Intel Xeon E5530 CPU with hyperthreading and 12 Gb of RAM, running Microsoft Windows XP Professional 2003 x64 Edition, Service Pack 2. To enable a more direct comparison between the command line and Oracle cartridge implementations, all searches were repeated in single-threaded mode on an HP blade equipped with 2 dual-core 3.60 GHz CPUs and 4 Gb of memory running 32-bit RedHat Enterprise Linux Server version 3.4.6–10 and Oracle Database Enterprise Edition

version 10.2.0.4. The results are illustrated in Figure 12. On average, Oracle took 29% longer to execute compared to the command line utility on the same hardware (HP), with the ratio of Oracle over command line total time ranging from 0.37 to 13.42 and having a median of 1.29. (The absolute times in Figure 12 are longer, as the HP blade was considerably slower than the IBM ThinkStation; on average, queries took 2.71 times longer to execute on the HP blade compared to the IBM ThinkStation using the command line utility with a single thread.)

CONCLUSIONS

We presented a fast and efficient substructure search algorithm that is well suited for both command-line and relational implementations. Our indexing strategy utilizes a new set of topological keys that make use of count information and show improved selectivity over established methods. Many path- and fragment-based fingerprints are not able to handle queries that contain nonspecific match points, such as [#7]-[#6], etc.; by contrast, our method was designed to handle such cases by generating keys derived from generic atom and bond descriptions. Our method is capable of completing a typical query of a million-member database in less than a second and is highly parallelizable on machines having multiple CPU and/or processing cores. The use of inverted bitsets results in very fast screening and the use of binary-encoded molecules (with pre-calculated molecular perception flags) results in faster verification. We also presented a scheme that enables the Oracle cost-based optimizer to construct efficient execution paths for complex queries combining substructure and other predicates. Our algorithm has been used successfully by hundreds of scientists within our discovery organization over a period of seven years. This exposure has taught us that search performance is critical for a positive end-user experience. As proprietary and public chemical databases grow larger in size, the need to search them efficiently will require not just more powerful hardware but also more intelligent search algorithms. We hope that this paper will rekindle interest in this important area of chemoinformatics research.

ASSOCIATED CONTENT

S Supporting Information. The usage (manpage) of the command line utility. This material is available free of charge via the Internet at <http://pubs.acs.org>.

AUTHOR INFORMATION

Corresponding Author

*Phone: (215) 628-6814. E-mail: dagrafio@its.jnj.com.

Present Addresses

[§]Autodesk Scaleform, Greenbelt, MD.

[¶]Jabiru Ventures L.L.C., New York, NY.

ACKNOWLEDGMENT

We would like to thank the hundreds of users of ABCD whose valuable feedback over the last seven years has led to many improvements in the performance and usability of our tools. Particular thanks are due to Dr. Peter J. Connolly who, in his role as Chair of the ABCD Scientific Advisory Team, has guided

much of this effort from the users' perspective and stewarded numerous user acceptance tests and software deployments. We also thank Dr. Mark Seierstad for useful discussions and performance tests during the development of the original algorithm and Dr. Renee DesJarlais for providing similar feedback and also for carefully reviewing this manuscript. Finally, we thank Dr. Wolf-Dietrich Ihlenfeldt for assistance with the CACTVS toolkit and Dr. Adel Golovin and Kim Henrick for their generous help and insightful communications on the search results.

REFERENCES

- (1) Friedman, T. L. *The world is flat: A brief history of the twenty-first century*; Farrar, Straus and Giroux: New York, 2005; p 157.
- (2) Weininger, D. SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules. *J. Chem. Inf. Comput. Sci.* **1988**, *28*, 31–36.
- (3) Dalby, A.; Nourse, J. G.; Hounshell, W. D.; Gushurst, A. K. I.; Grier, D. L.; Leland, B. A.; Laufer, J. Description of several chemical structure file formats used by computer programs developed at Molecular Design Limited. *J. Chem. Inf. Comput. Sci.* **1992**, *32*, 244–255.
- (4) Willett, P. A review of chemical structure retrieval systems. *J. Chemom.* **1987**, *1*, 139–155.
- (5) Barnard, J. M. Substructure searching methods: old and new. *J. Chem. Inf. Comput. Sci.* **1993**, *33*, 532–538.
- (6) Attias, R.; Dubois, J.-E. Substructure systems: concepts and classifications. *J. Chem. Inf. Comput. Sci.* **1990**, *30*, 2–7.
- (7) Ray, L. C.; Kirsch, R. A. Finding chemical records by digital computers. *Science* **1957**, *126*, 814–819.
- (8) Xu, J.; Zhang, M. HBA: new algorithm for structural match and applications. *Tetrahedron Comput. Methodol.* **1989**, *2*, 75–83.
- (9) Dengler, A.; Ugi, I. A central atom based algorithm and computer program for substructure search. *Comput. Chem.* **1991**, *15*, 103–107.
- (10) Xu, J. GMA: a generic match algorithm for structural homorphism, isomorphism, maximal common substructure match and its applications. *J. Chem. Inf. Comput. Sci.* **1996**, *36*, 25–34.
- (11) Sussenguth, E. H., Jr. A graph-theoretic algorithm for matching chemical structures. *J. Chem. Doc.* **1965**, *5*, 36–43.
- (12) Figueras, J. Substructure search by set reduction. *J. Chem. Doc.* **1972**, *12*, 237–244.
- (13) Ullmann, J. R. An algorithm for subgraph isomorphism. *J. Assoc. Comput. Mach.* **1976**, *23*, 31–42.
- (14) Von Scholley, A. A relaxation algorithm for generic chemical structure screening. *J. Chem. Inf. Comput. Sci.* **1984**, *24*, 235–241.
- (15) Lynch, M. F. Screening large chemical files. In *Chemical information systems*; Ash, J. E., Hyde, E., Eds.; Ellis Horwood: Chichester, 1974; pp 177–194.
- (16) Wiswesser, W. J. Historic development of chemical notations. *J. Chem. Inf. Comput. Sci.* **1985**, *25*, 258–263.
- (17) Thomson, L. H.; Hyde, E.; Matthews, F. W. Organic search and display using a connectivity matrix derived from Wiswesser notation. *J. Chem. Doc.* **1967**, *7*, 204–209.
- (18) Attias, R. DARC substructure search system: a new approach to chemical information. *J. Chem. Inf. Comput. Sci.* **1983**, *23*, 102–108.
- (19) Bremser, W. HOSE— a novel substructure code. *Anal. Chim. Acta* **1978**, *103*, 355–365.
- (20) Fisanick, W. The Chemical Abstracts Service generic chemical (Markush) structure storage and retrieval capability. 1. Basic concepts. *J. Chem. Inf. Comput. Sci.* **1990**, *30*, 145–155.
- (21) Shenton, K.; Norton, P.; Fearn, E. A. Generic searching of patent information. In *Chemical Structures - the International Language of Chemistry*; Warr, W. A., Ed.; Springer: Berlin, 1988; pp 169–178.
- (22) Feldmann, R. J.; Milne, G. W. A.; Heller, S. R.; Fein, A.; Miller, J. A.; Koch, B. An interactive substructure search system. *J. Chem. Inf. Comput. Sci.* **1977**, *17*, 157–163.
- (23) Bruck, P.; Nagy, M. Z.; Kozics, S. Substructure search on hierarchical trees. In *Proceedings of the 11th International Online Information Meeting*; Learned Information: Oxford, London, 1987; Vol. 87, pp 41–43.
- (24) Hicks, M. G.; Jochum, C. Substructure search systems. 1. Performance comparison of the MACCS, DARC, HTSS, CAS Registry MVSSS, and S4 substructure search systems. *J. Chem. Inf. Comput. Sci.* **1990**, *30*, 191–199.
- (25) Ozawa, K.; Yasuda, T.; Fujita, S. Substructure search with tree-structured data. *J. Chem. Inf. Comput. Sci.* **1997**, *37*, 688–695.
- (26) Pavlov, D.; Shturms, I. Chemical substructure search screening with fingerprints built with subgraph enumeration. *Rev. Adv. Mater. Sci.* **2009**, *20*, 37–41.
- (27) Todeschini, R.; Consonni, V. *Handbook of Molecular Descriptors*; Wiley-VCH: Weinheim, Germany, 2000.
- (28) Nikolova, N.; Jaworska, J. Approaches to measure chemical similarity - a review. *QSAR Comb. Sci.* **2003**, *22*, 1006–1026.
- (29) <http://www.daylight.com/dayhtml/doc/theory/theory.finger.html> (accessed June 22, 2011).
- (30) Baldi, P.; Benz, R. W.; Hirschberg, D. S.; Swamidass, S. J. Lossless compression of chemical fingerprints using integer entropy codes improves storage and retrieval. *J. Chem. Inf. Model.* **2007**, *47*, 2098–2109.
- (31) Golovin, A.; Henrick, K. Chemical substructure search in SQL. *J. Chem. Inf. Model.* **2009**, *49*, 22–27.
- (32) Agrafiotis, D. K.; Alex, S.; Dai, H.; Derkinderen, A.; Farnum, M.; Gates, P.; Izrailev, S.; Jaeger, E. P.; Konstant, P.; Leung, A.; Lobanov, V. S.; Marichal, P.; Martin, D.; Rassokhin, D. N.; Shemanarev, M.; Skalkin, A.; Stong, J.; Tabruyn, T.; Vermeiren, M.; Wan, J.; Xu, X. Y.; Yao, X. Advanced Biological and Chemical Discovery (ABCD): centralizing discovery knowledge in an inherently decentralized world. *J. Chem. Inf. Model.* **2007**, *47*, 1999–2014.
- (33) Kirkpatrick, P. Informatics: the ABCD of data management. *Nat. Rev. Drug Discovery* **2007**, *6*, 956–957.
- (34) Swamidass, S. J.; Baldi, P. Bounds and algorithms for fast exact searches of chemical fingerprints in linear and sublinear time. *J. Chem. Inf. Model.* **2007**, *47*, 302–317.
- (35) Swamidass, S. J.; Chen, J.; Bruand, J.; Phung, P.; Ralaivola, L.; Baldi, P. Kernels for small molecules and the prediction of mutagenicity, toxicity and anti-cancer activity. *Bioinformatics* **2005**, *21*, 359–368.
- (36) Ramabadrhan, T. V.; Gaitonde, S. S. A tutorial on CRC computations. *IEEE Micro* **1988**, *8* (4), 62–75.
- (37) Daylight Chemical Information Systems Inc. <http://www.daylight.com/dayhtml/doc/theory/theory.smarts.html#RTFTtoC35> (accessed June 1, 2011).
- (38) Agrafiotis, D. K.; Bandyopadhyay, D.; Farnum, M. Radial clustergrams: visualizing the aggregate properties of hierarchical clusters. *J. Chem. Inf. Model.* **2007**, *47*, 69–75.
- (39) Cepeda, M. S.; Lobanov, V. S.; Farnum, M.; Weinstein, R.; Gates, P.; Agrafiotis, D. K.; Stang, P.; Berlin, J. A. Broadening access to electronic health care databases. *Nat. Rev. Drug Discovery* **2010**, *9*, 84.
- (40) Agrafiotis, D. K.; Shemanarev, M.; Connolly, P. J.; Farnum, M.; Lobanov, V. S. SAR maps: a new SAR visualization technique for medicinal chemists. *J. Med. Chem.* **2007**, *50*, 5926–5937.
- (41) Kolpak, J.; Connolly, P. J.; Lobanov, V. S.; Agrafiotis, D. K. Enhanced SAR maps: Expanding the data rendering capabilities of a popular medicinal chemistry tool. *J. Chem. Inf. Model.* **2009**, *49*, 2221–2230.
- (42) Agrafiotis, D. K.; Wiener, J. J. M. Scaffold Explorer: An interactive tool for organizing and mining SAR data spanning multiple chemotypes. *J. Med. Chem.* **2010**, *53*, S002–S011.
- (43) Agrafiotis, D. K.; Wiener, J. J. M.; Skalkin, A.; Kolpak, J. Single R-group polymorphisms (SRPs) and R-cliffs: An intuitive framework for analyzing and visualizing activity cliffs in a single analog series. *J. Chem. Inf. Model.* **2011**, *51*, 1122–1131.
- (44) Altschul, S. F.; Gish, W.; Miller, W.; Myers, E. W.; Lipman, D. J. Basic local alignment search tool. *J. Mol. Biol.* **1990**, *215* (3), 403–410.
- (45) Agrafiotis, D. K.; Lobanov, V. S.; Salemme, F. R. Combinatorial informatics in the post-genomics era. *Nat. Rev. Drug Discovery* **2002**, *1*, 337–346.

- (46) Shemanarev, M. The Anti-Grain Geometry Project. <http://www.antigrain.com> (accessed June 22, 2011).
- (47) <http://www.symyx.com>. (accessed June 22, 2011).
- (48) <http://www.cambridgesoft.com>. (accessed June 22, 2011).
- (49) McGregor, M. J.; Pallai, P. V. Clustering of large databases of compounds using MDL keys as structural descriptors. *J. Chem. Inf. Comput. Sci.* **1997**, *37*, 443–448.
- (50) Durant, J. L.; Leland, B. A.; Henry, D. R.; Nourse, J. G. Reoptimization of MDL keys for use in drug discovery. *J. Chem. Inf. Comput. Sci.* **2002**, *46*, 1273–1280.
- (51) Ihlenfeldt, W. D.; Takahashi, Y.; Abe, H.; Sasaki, S. Computation and management of chemical properties in CACTVS: An extensible networked approach toward modularity and compatibility. *J. Chem. Inf. Comput. Sci.* **1994**, *34*, 109–116.
- (52) <http://openbabel.org/wiki/FP2>. (accessed June 22, 2011).
- (53) Guha, R.; Howard, M. T.; Hutchison, G. R.; Murray-Rust, P.; Rzepa, H.; Steinbeck, C.; Wegner, J. K.; Willighagen, E. The Blue Obelisk - Interoperability in chemical informatics. *J. Chem. Inf. Model.* **2006**, *46*, 991–998.