Fast Similarity Searches and Similarity Joins in Oracle DB

Astrid Rheinländer, Ulf Leser
Humboldt-Universität zu Berlin
Wissensmanagement in der Bioinformatik

Keywords:

Similarity Search, Similarity Join, Prefix Tree Index, Extensible Indexing Interface

Introduction

Similarity search and similarity join on strings are important operations for applications such as duplicate detection, error detection, data cleansing, or comparison of biological sequences [GIJ+01, NMS04]. Especially DNA sequencing produces large collections of erroneous strings which need to be searched, compared, and merged. In our talk, we will use ESTs as our running example. ESTs (Expressed Sequence Tags) are short DNA sequences with lengths mostly in the region of 300 to 800 bases that are commonly used to identify genes and their localization on a chromosome. However, to be cost-effective, ESTs are obtained by a single sequencing pass which yields in an estimated error rate of 1% [KA06]. This implies that searching and joining EST data sets should always be carried out approximately rather than exactly. Since the EST sets that are considered go in the millions , efficient execution of such similarity operations is crucial.

Oracle DB has no build-in support for similarity searching. A naive implementation using stored procedures does not scale to the amount of data produced in Life Science projects (see section "Experiments"). We developed PETER, a prefix tree based indexing algorithm supporting approximate search and approximate joins. Our tool supports Hamming and edit distance as similarity measure and is available as a cartridge for Oracle DB, as C++ library, and as command line tool. It combines an efficient implementation of compressed prefix trees with advanced pre-filtering techniques that exclude many candidate strings early. We evaluate our tool on several collections of long strings containing up to 5,000,000 entries of length up to 3,500. Our experiments reveal that PETER is faster by orders of magnitude compared to Oracle in the inexact case. We also show that PETER outperforms the built-in join methods on such data sets (very long strings) even in the exact case.

A technical description of our method appeared in [RKH+10]. In this talk, we concentrate on issues regarding the integration of our method into the Oracle database.

Similarity Measures

Similarity-based operations must be based on a concrete similarity measure. PETER supports:

Hamming distance
The Hamming distance $d_{hd}(s, t)$ of two strings s, t of equal length is the number of mismatching characters in s and t. We say two strings are within Hamming distance k if $d_{hd}(s, t) \leq k$. Obviously, computing the Hamming distance of two strings with $|s| = |t| = n$ is possible in $O(n)$.

Edit distance:
The edit distance $d_{ed}(s1, s2)$ of two strings *s*, *t* with $|s| = n$, $|t| = m$ is the minimal number of insertions, deletions, or replacements of single characters needed to transfrom *s* into *t.* We say two strings are within edit distance *k*, if $d_{ed}(s, t) \leq k$. Using dynamic programming, the edit distance can be computed $O(m*n)$ [G01]. However, faster computation is possible when one is only interested in highly similar strings. The *k*-banded alignment algorithm [F84] finds the edit distance of two strings with edit distance of at most $2k$ in $O(k * max\{m, n\})$.

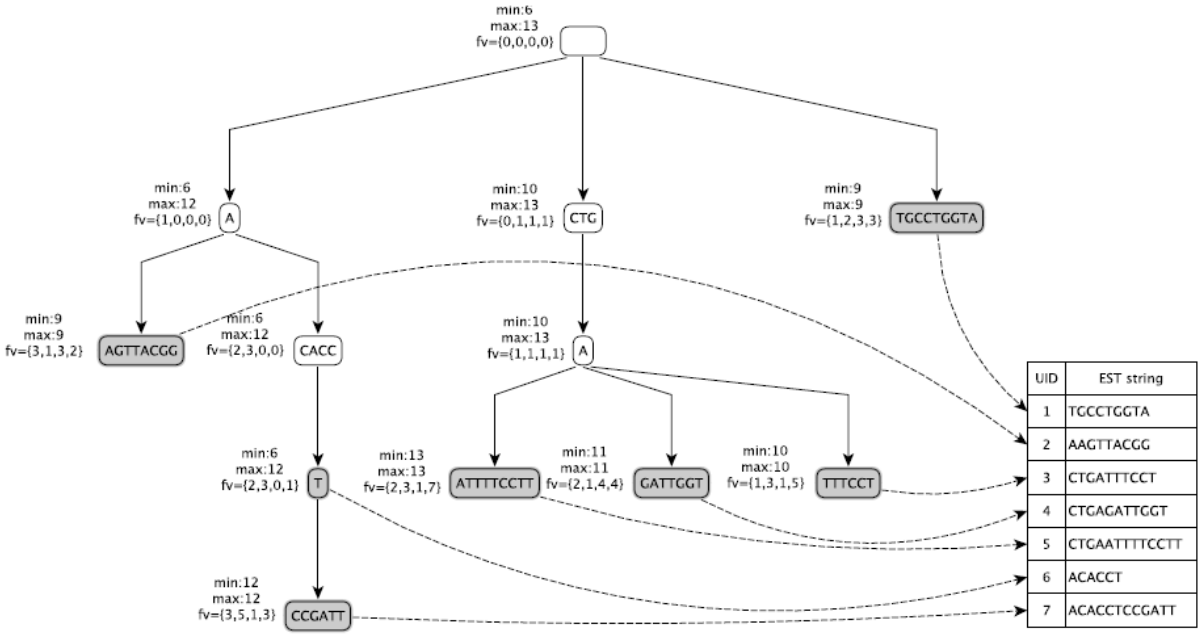

Fig. 1: Compressed prefix tree. Grey nodes are string nodes. Min/max specify minimum and maximum string lengths, fv denotes the frequency vector.

Compressed Prefix Trees
Our fundamental data structure are compressed prefix trees [D68], built on top of a set of strings.
A compressed prefix tree index *T* for a set of strings *R* is a rooted, directed tree that meets the following conditions:

1. Every node *x* is labeled with a sequence of characters $c_i \in \Sigma$ of length $l \geq 1$. The labels of any two children *y, z* of the same node *x* start with a different character.
2. Every string $s \in R$ maps to some node $x \in$ T such that the concatenation of all labels from *T*'s root to *x* exactly is *s*. We call *x* string node and assign the corresponding ID to *x*.
3. (Compression of suffixes). Let *x* be the root of a subtree formed by a linear chain of children $x_1,...,x_m$, where solely x_m is a string node and has no further children. Then, $x,x_1,...,x_m$ are merged to a single node *x'* whose label is the concatenation of their labels. The ID of x_m is assigned to *x'*.
4. (Compression of infixes). Let *x* be the root of a subtree formed by a linear chain of children $x_1,...,x_m$, where no node is a string node and only x_m has more than one child. Then, $x,x_1,...,x_{m-1}$ are merged to a single node *x'* whose label is the concatenation of their labels.

We attach further information to every node, namely minimum/maximum string lengths and a frequency vector. Figure 1 shows an example of a compressed prefix tree. It has simple nodes (e.g., "A"), a compressed infix node ("CTG"), and a compressed suffix node ("TGCCTGGTA").

Algorithms

Search
PETER essentially performs a depth-first traversal of the prefix tree applying different pruning techniques [RKH+10] which will be explained in detail in our talk. Before descending from a node, we apply length and frequency filters. We also prune if we exceed the allowed distance threshold. In case of edit distance searches, we apply *q*-gram filtering for every reached string node PETER checks a flag whether a Hamming or edit distance search is performed and computes the new distance. When a match was found, the pair of matching EST objects (each pair consists of the ESTs and their UIDs) is added to the result set. Finally, the result set is returned to the user and by default printed to `stdout`.

Join
When using the join operator on two relations, conceptionally the intersection of two trees is computed. Both index trees *T* and *S* are traversed concurrently, such that the tree with less nodes is traversed first. Tree sizes are checked at startup. Length and frequency filtering are applied as in the search algorithm. If we reach a string node in tree *T* (or *S*, respectively), we fetch the complete string represented by this node and perform a call to the search function, with the string as pattern, the remaining subtree *S′* of *S (T′* of *T)* and the current distance value as parameters. We continue the distance computation for the next untreated characters in the string and *S′ (T′).* The result set contains all pairs of matching EST objects and is constructed through the search algorithm. Finally, it is returned to the user and printed to `stdout`.

Integration in Oracle DB

We integrated our data structure and algorithms in Oracle 10g XE as a shared library using the Oracle Data Cartridge Interface (ODCI). Amongst others, the ODCI provides functionality to user-defined indexes (via Extensible Indexing Interface) and table functions (Table Function Interface). Integrating user-defined functions and indexes in Oracle consists of two parts: The first part is the program code, compiled as a shared library outside of Oracle and saved in the `$ORA_HOME/bin` directory. The second part involves declarations and definitions in PL/SQL directly executed in Oracle, including a reference to
the library. When index and query functions are accessed for the first time in a session, PL/SQL uses the data dictionary to determine the location of the shared library in the file system. A listener process immediately invokes a session-specific agent (`extproc`) and passes the call including procedure and library name and any parameters, if present, to it. `Extproc` then loads the library and runs the desired function, that, in our case, in turn opens the index and later on suffix files if required. Any return values are passed back via `extproc` to PL/SQL. Throughout the session, `extproc` remains alive, which implies that initialization costs for `extproc` emerge only once. Figure 2 shows the interplay of Oracle and prefix tree index components in control flow.

Experiments

We use ESTs to evaluate the performance of PETER both for similarity and for exact operations *(k=0).* Index creation and optimization was performed in advance and is not included in the measured times (see Fig.3). We observed that the time for index creation grows, as expected, linear with the number of indexed strings. We compare the performance of PETER against two competitors: The Unix tools command line tools grep, agrep, and nrgrep, and build-in or user-defined functions (UDF) inside Oracle. Our evaluation revealed that a combination of length and *q*-gram filtering seems to be the best overall configuration. Therefore, in all following experiments with PETER we always used

length filtering for Hamming distance and a combination of length and *q*-gram filtering for edit distance searches and joins.

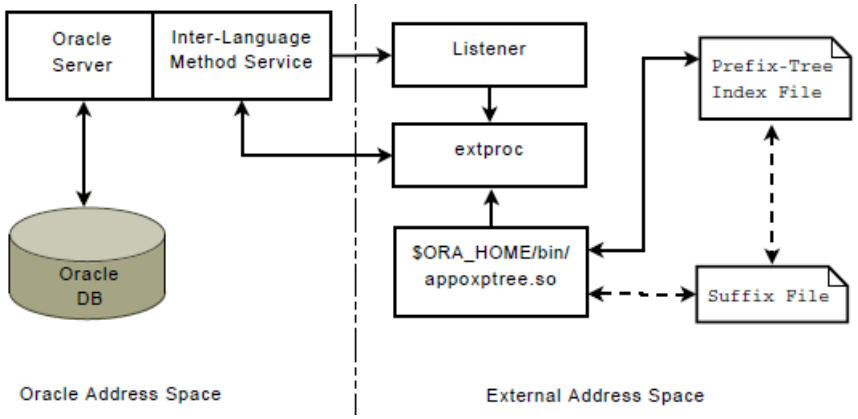

Fig. 2: Execution of PETER operators inside Oracle DB.

Performance of Similarity Search and Joins

We compared the execution times of PETER for Hamming and edit distance for various thresholds to Unix command line tools. We used grep for $k = 0$, and agrep and nrgrep for $k \epsilon \{1,2,3,8\}$, respectively. First, we performed individual searches for each pattern $p \epsilon T_3$ in the indexed EST set T_1. When searching with the Unix tools all searches were started to match only complete strings to the given pattern. For exact search, we outperform grep significantly with a factor of 63 for Hamming distance and a factor of 50 for edit distance scoring enabled. Figure 4 contrasts the average execution times of inexact searches to agrep and nrgrep. For very short patterns, we outperform agrep with factors in the range of 640 for $k = 1$ up to 1063 for $k = 8$ on Hamming distance and a factor up to 450 for edit distance constraints. When searching with patterns of arbitrary length, we are up to 60 times faster than nrgrep for Hamming distance and up to 45 times faster for edit distance. Even if we add the costs for index creation to the evaluation, PETER amortizes quite fast. For example, if we run multiple Hamming distance (edit distance, respectively) searches in T_1 with $k = 1$, it takes only 10 (15) searches to outperform the cumulated runtimes of agrep. Compared to nrgrep, it takes 125 (105) Hamming distance (edit distance) queries until PETER is profitable.

Set	# EST strings	avg. string length	min/max length	# tree nodes	# ext. suffixes
T_1	307,542	348	14/3,615	589,062	293,764
T_2	736,305	387	12/3,707	1,482,709	689,590
T_{2a}	368,152	382	12/2,774	711,632	352,872
T_{2b}	184,076	385	22/2,774	349,329	177,846
T_{2c}	92,038	383	25/2,774	171,964	89,198
T_{2d}	46,019	381	28/2,774	84,954	44,716
T_{2e}	23,009	373	31/ 878	42,375	22,366
T_3	10,000	536	16/3,707	16,310	8,774
TX	5,000,000	359	14/3,247	10,478,214	4,834,231

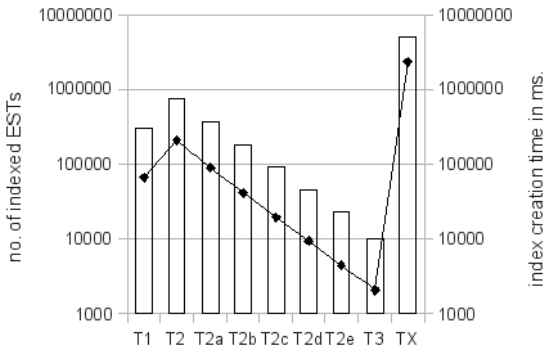

Fig. 3. Left: Properties of EST sets. Right: Index creation (line) wrt. set size (bar).

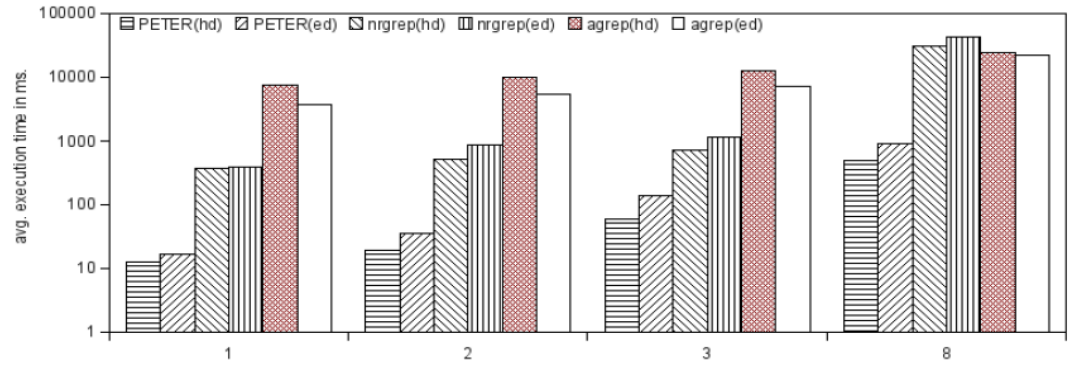

Fig. 4. Search in PETER vs. Unix tools for $p \epsilon T_3$, $k \epsilon \{1,2,3,8\}$ *(log-scale).*

For approximate joins, we are not aware of any Unix command line tool that could handle this problem. Comparing edit distance to Hamming distance joins, the latter always performed in a range of 30% to 60 % better, mostly dependent on the given threshold. We observe an exponential growth of join execution times with respect to the threshold although the result sets don't grow exponentially. The reason for this is that the search space increases exponentially with growing *k*. While tree traversal, PETER descends further as k grows and for every additional node, that is reached in T , there are $|\sigma|$ additional sub-trees examined in *S*.

Performance inside Oracle

We compared PETER's performance against exact and similarity-based search and joins inside the Oracle. For searching, we performed single `SELECT` queries on the B*-indexed relation T_1 for each EST string in T_3. At all times and for different pattern length, the built-in `SELECT`-operator achieves better runtimes than a prefix tree based search. Factors vary dependent on the pattern length, in a range of 2 $(|p| \leq 400)$ to 1.3 $(|p| \geq 800)$. There are mostly two reasons for this result. First, the operations in the prefix tree index are handled via the extension interface which produces overhead for every call. Second, the extension interface does not allow caching of data. While the internal implementation uses the internal buffer pool of the database to cache the most important parts of the B*-index, this is not possible for user-defined indexing. Given these severe drawbacks, it is notably that PETER is only so little slower.

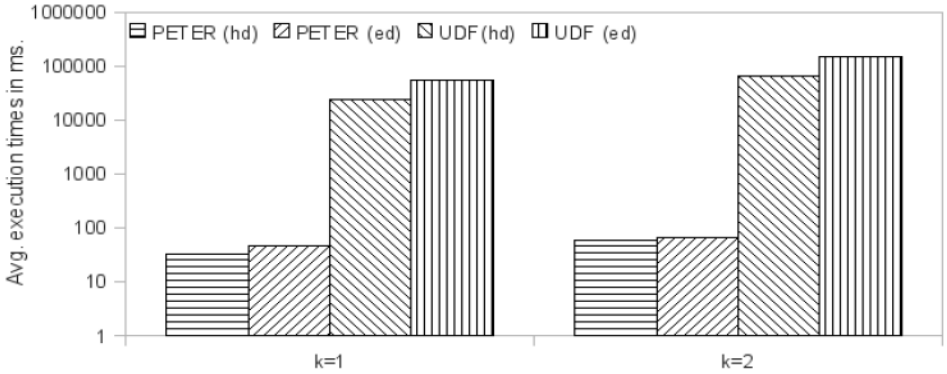

Fig. 5. Comparison of PETER with UDF searches (log-scale).

Regarding joins, we computed $T_1 \bowtie T_{2i}$ as a Hash join and as a Sort-Merge join and compared these results to PETER. Joins on the prefix tree index always outperformed both Hash join (with factors between 1.5 and 4) and Sort-Merge join (with factors 3.8 to 10). Note that the problem of caching is not a severe one here, as computing the join requires loading both indexes only once.

As there are no built-in functions for similarity operations inside Oracle DB, we implemented them as user-defined functions (UDF) in PL/SQL. We compared the execution time of UDF-based similarity search and joins to PETER. Fig.5 shows that PETER for similarity searches performs better by an order of magnitude than using just UDFs. For Hamming distance search, prefix tree indexing leads to a runtime improvement factor of about 520, for edit distance searches of about 890. We also tried to perform similarity joins for $k = 1$ with UDFs on $T_3 \bowtie T_{2e}$, but as the join operations did not finish within a day, we aborted the execution. Similarity joins with prefix trees finished for $k \epsilon \{1,2\}$ in less than one minute.

References

[D68] D. R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. Journal of the ACM (JACM), 15(4), 1968.

[F84] J. W. Fickett. Fast optimal alignment. Nucleic Acids Research, 12, 1984.

[G01] G. Navarro. A guided tour to approximate string matching. ACM Computing Surveys, 33(1), 2001.

[GIJ+01] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In VLDB 2001.

[KA06] A. Kalyanaraman and S. Alaru. Expressed sequence tags: Clustering and applications. In Handbook of Computational Molecular Biology, Boca Raton, 2006. Chapman & Hall / CRC computer information science.

[NMS04] N. Koudas, A. Marathe, and D. Srivastava. Flexible string matching against large databases in practice. In VLDB 2004.

[RKH+10] A. Rheinländer, M. Knobloch, N. Hochmuth, and U. Leser. Prefix Tree Indexing for Similarity Search and Similarity Join on Genomic Data. In SSDBM 2010.

Contact:

Astrid Rheinländer, Ulf Leser
Humboldt-Universität zu Berlin
Institut für Informatik
Wissensmanagement in der Bioinformatik
Unter den Linden 6
D-10099 Berlin

E-Mail {rheinlae,leser}@informatik.hu-berlin.de
Internet: http://www.informatik.hu-berlin.de/wbi