APPLICATIONS OF THE DYNAMIC N-DIMENSIONAL K-VECTOR

Carl Leake,* Javier Roa,† and Daniele Mortari‡

The n-dimensional k-vector (NDKV) is an appealing alternative to binary tress for resolving complex queries in large relational databases. The method has excelled in several applications involving static databases. The present paper extends the theory supporting the NDKV to handle dynamic databases, where the data is updated frequently. This includes deleting records, adding new entries, or editing existing elements. The merit of this new version of the NDKV, the dynamic n-dimensional k-vector (DNDKV), is that it is no longer necessary to recompute the entire k-vector (the main structure that indexes the data) every time a record changes. The algorithm updates the four constituents of the standard NDKV on the fly: the database, sorted database, index, and k-vector tables. As a result, the DNDKV becomes comparable in terms of capabilities and flexibility to state-of-the-art storage engines relying on structured query languages (SQL). The performance of the DNDKV is assessed by running typical read/write operations on a database that contains millions of pre-computed missions to celestial bodies. This database requires frequent updates whenever an orbit solution is refined or new bodies are discovered. The DNDKV is faster than rebuilding the k-vector tables completely, provided that the number of elements being added or removed is not excessively large. Direct runtime comparisons with MySQL suggest that the DNDKV is several times faster for reading but might be slower for writing and updating the database. One limit of the technique is the elements being added must be within the range of the current k-vector tables. If this is not the case, the technique cannot be used and the k-vector tables must be rebuilt from scratch.

INTRODUCTION

For two decades, the k-vector range searching algorithm has been used to solve the Star-ID problem, generating the state-of-the-art Star-ID algorithm, Pyramid [1]. More recently, the k-vector has been used in a new star identification algorithm, Super k-ID, which won the "Star Trackers: First Contact" competition, an ESA Advanced Concepts competition [2]. In addition, the k-vector was successfully applied to solve a set of new problems, including interpolation and inversion of nonlinear functions [3].

The main idea of the k-vector is to describe the nonlinearities of a sorted database. The key feature of the k-vector is that the searching time is independent from the database size. However, the search time depends on how nonlinear the sorted database is. Using the adaptive k-vector [4], more memory is allocated to improve k-vector performance when data is strongly nonlinear. Reference [5] provides an initial description and analysis of this searching method for one-dimensional databases.

*Graduate Student, Aerospace Engineering, Texas A&M University, College Station, TX. E-mail: LEAKEC@TAMU.EDU

†Navigation Engineer, Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, E-mail: JAVIER.ROA@JPL.NASA.GOV

‡Aerospace Engineering, Texas A&M University, College Station, TX. IEEE and AAS Fellow, AIAA Associate Fellow. E-mail: MORTARI@TAMU.EDU

Recently [6], the one-dimensional k-vector was extended to n-dimensions; it is called the n-dimensional k-vector (NDKV). The NDKV has been used to solve problems such as: massive multidimensional database searching, satellite coverage, and observer position estimation using pulsars. However, all of these problems use static databases.

This paper extends the use of NDKV by providing a technique to handle dynamic databases, and applies that technique to a database of missions to celestial bodies. Extending the NDKV to dynamic databases increases the utility of the NDKV to astrodynamicists because many astrodynamic databases –databases of orbital debris, databases of trajectory data, databases of potential missions, etc.– are dynamic. This technique avoids completely rebuilding the n k-vector tables by leveraging some of their properties to efficiently update them. Thus, avoiding the computationally expensive process of rebuilding the n k-vector tables from scratch.

The paper is organized as follows. The next section introduces the concept of the NDKV and some key definitions. Next, the two sections that follow explain how the dynamic NDKV (DNDKV) is constructed by adding and removing elements from the database. After discussing the potential for parallelization, the database of pre-computed missions used for testing is defined. The last section includes the tests that compare the performance of the DNDKV with MySQL, a very common database management service used in production systems.

PARTS OF THE NDKV

Before delving into the technique to handle dynamic databases, it is important to understand the constituents of the NDKV. Note, this section assumes some familiarity with the general NDKV technique. Readers lacking familiarity with the general technique may refer to the 1-*dimensional k-vector* and *NDKV* sections of Ref. [6].

The NDKV consists of four constituents*:

1. database
2. sorted database (sorted row-wise)
3. index that maps from the database to the sorted database (referred to hereafter as the index)
4. k-vector tables

All of these constituents are stored as a matrix of size $[N_D, N_E]$ where N_D is the number of dimensions and N_E is the number of elements. The mathematical relationship between the first three constituents is shown in Eq. (1),

$$\mathbf{S}_i(j) = \mathbf{D}_i(\mathbf{I}_i(j)) \quad \text{where} \quad i \in [1, N_D], \tag{1}$$

$\mathbf{S}$ is the sorted database, $\mathbf{D}$ is the database, and $\mathbf{I}$ is the index.

The fourth constituent of the NDKV is the k-vector tables. For readers who are unfamiliar with how the k-vector tables are constructed and used, refer to Ref. [6]. Restated here, because it is of significant importance for adding and deleting elements in the database, is how to retrieve elements between two indices, $[y_a, y_b]$, along one of the dimensions. Let i indicate the dimension being

*The NDKV may instead consist of three constituents where the sorted database is removed from the set and the sorted database is "accessed" via the database and the index as shown in Eq. (1)

searched along, m_i be the slope of the k-vector for the given dimension, and q_i be the intercept of the k-vector for the given dimension. Two indices, k_a and k_b, are defined according to

$$k_a = \left\lfloor \frac{y_a - q}{m} \right\rfloor + 1, \qquad \text{and} \qquad k_b = \left\lceil \frac{y_b - q}{m} \right\rceil, \tag{2}$$

where $\lfloor X \rfloor$ is the greatest integer lower than X (the `floor` function), and $\lceil X \rceil$ is the lowest integer greater than X (the `ceil` function). The elements $\mathbf{D}_i(j)$ that are within the search range $[y_a, y_b]$ are $\mathbf{D}_i(\mathbf{I}_i(\mathbf{K}_i(k_a : k_b)))$, or as indexed via MATLAB would be $\mathbf{D}(i, \mathbf{I}(i, \mathbf{K}(i, k_a : k_b)))$, where $\mathbf{K}$ is the k-vector table. It should be noted that k_a and k_b may return extraneous elements; therefore, a very small linear search may need to be performed at the endpoints to ensure no extraneous elements exist.

The following two sections demonstrate how to efficiently add and remove elements from the NDKV. Each of these processes will require updating the four constituents of the NDKV. When programming the process, the authors found it easiest and most efficient to implement the technique via a class structure. Using the class structure made it simple to internally add features, such as padding arrays with extra zeros to avoid reallocating memory each time an element was added, that may be more difficult to handle with a set of independent functions. Furthermore, each of these processes will require looping over the number of dimensions. Therefore, for a 1-dimensional k-vector (or just k-vector) this loop can be removed.

ADDING ELEMENTS

Adding elements to the database requires updating the four constituents of the NDKV. First, a double loop is implemented that updates the database, index, and k-vector tables. The outer loop is over the dimension and the inner loop is over the elements that are being added. The outer loop will use index i where $i \in [1, N_D]$ and the inner loop will use index j where $j \in [1, N_A]$ where N_A is the number of elements being added. Then, the sorted database is rebuilt by using the database and the index. For brevity, a MATLAB style convention will be used in this section for indexing matrices. For example, the element in the i^{th} row and the j^{th} column of matrix $\mathbf{A}$ would be accessed by $\mathbf{A}(i, j)$. In addition, a colon will indicate all possible index values along a dimension. For example, the entire first row of $\mathbf{A}$ can be indexed using $\mathbf{A}(1, :)$.

The index and k-vector tables are updated by leveraging two properties of the k-vector tables. The first property is the values of the k-vector tables indicate how many data points are below the associated value in the mapping function. For example, if $\mathbf{K}(i, 3)$ is 5, then there are 5 elements below the third value in the mapping function. The second property is the k-vector tables can be used to quickly locate the position in the index where the new element will be added.

Thanks to the first property of the k-vector, it is possible to locate the positions in the k-vector tables where the changes need to be made:

$$\begin{aligned} K_{\text{ind}} &= \left\lceil \frac{\mathbf{E}_A(i, j)}{m(i)} + q(i) \right\rceil \\ \mathbf{K}(:, K_{\text{ind}} : N_E) &= \mathbf{K}(:, K_{\text{ind}} : N_E) + 1 \end{aligned} \tag{3}$$

where $\mathbf{E}_A$ is the set of column vector elements to be added. Equation (3) updates the k-vector tables.

The position in the index where the new element will be stored is located making use of the second property,

$$k_b = \mathbf{K}(i, K_{\text{ind}} - 1), \tag{4}$$

where k_b is the position in the index just below the position where the new element will be inserted and K_{ind} is found according to Eq. (3). A linear search is used to increment the value of k_b until the position of the index just below where the new element will be inserted is found. For databases that are nearly linear when sorted, this linear search typically has to go through 0 or 1 iterations to locate the position where the new element index will be inserted. The new element index is $N_E + j$. Then, the i^{th} row of the index can be restructured using

$$\mathbf{I}(i, 1 : N_E) = [\mathbf{I}(i, 1 : kb), N_E + j, \mathbf{I}(i, kb + 1 : N_E)] \tag{5}$$

and the database is updated inside the double loop like

$$\mathbf{D}(i, N_E + j) = \mathbf{E}_{\mathbf{A}}(i, j). \tag{6}$$

Once the double loop is finished, the final steps are to update the number of elements and rebuild the sorted database. The number of elements N_E is updated with the number of added elements, N_A,

$$N_E = N_E + N_A. \tag{7}$$

Finally, the sorted database is rebuilt row by row according to

$$\mathbf{S}(i, 1 : N_E) = \mathbf{D}(i, \mathbf{I}(i, 1 : N_E)). \tag{8}$$

Algorithm 1 provides pseudocode that supplements the explanation given in this section.

Algorithm 1 Adding Elements

for $i \in [1, N_D]$ **do**
  for $j \in [1, N_A]$ **do**
    $K_{\text{ind}} = \texttt{ceil}(\mathbf{E}_A(i, j)/m(i) + q(i))$
    $k_b = \mathbf{K}(i, K_{\text{ind}} - 1)$
    {Perform a linear search to find the exact value of k_b}
    while $\mathbf{D}(i, \mathbf{I}(i, k_b)) > \mathbf{E}_A(i, j)$ **do**
      $k_b = k_b + 1$
    end while
    $\mathbf{K}(:, K_{\text{ind}} : N_E) = \mathbf{K}(:, K_{\text{ind}} : N_E) + 1$
    $\mathbf{I}(i, 1 : N_E) = [\mathbf{I}(i, 1 : kb), N_E + j, \mathbf{I}(i, kb + 1 : N_E)]$
    $\mathbf{D}(i, N_E + j) = \mathbf{E}_{\mathbf{A}}(i, j)$
  end for
end for
$N_E = N_E + N_A$
$\mathbf{S}(i, 1 : N_E) = \mathbf{D}(i, \mathbf{I}(i, 1 : N_E))$

REMOVING ELEMENTS

Removing elements from the database requires updating the four constituents of the NDKV. First, a double loop is implemented that updates the database, index, and k-vector tables. The outer loop is over the dimension and the inner loop is over the elements that are being removed. The outer loop will use index i where $i \in [1, N_D]$ and the inner loop will use index j where $j \in [1, N_R]$ where N_R is the number of elements being removed. Then, the sorted database is rebuilt by using the database and the index. As in the previous section, a MATLAB style convention will be used for indexing matrices.

The index and k-vector tables are updated by leveraging the two properties highlighted in the previous section. The first property is used to locate the positions in the k-vector tables where the changes need to be made:

$$\begin{aligned} K_{\text{ind}} &= \left\lceil \frac{\mathbf{E}_R(i,j)}{m(i)} + q(i) \right\rceil \\ \mathbf{K}(:, K_{\text{ind}} : N_E) &= \mathbf{K}(:, K_{\text{ind}} : N_E) - 1, \end{aligned} \tag{9}$$

where $\mathbf{E}_R$ is the set of column vector elements to be removed. Equation (9) updates the k-vector tables.

Equation (10) shows how the second property is used to locate the position in the index where the element to be removed is currently stored,

$$k_b = \mathbf{K}(i, K_{\text{ind}} - 1), \tag{10}$$

where k_b is the position in the index at or just below the position where the element to be removed is stored and K_{ind} is found according to Eq. (9). A linear search is used to increment the value of k_b until the position of the index where the element to be removed is stored is found. For databases that are nearly linear when sorted, this linear search typically has to go through 0 or 1 iterations to locate the position where the element to be removed is located. Then, the i^{th} row of the index can be restructured using

$$\mathbf{I}(i, 1 : N_E) = [\mathbf{I}(i, 1 : k_b - 1), \mathbf{I}(i, k_b + 1 : N_E), 0]. \tag{11}$$

A zero is added to the end of the index to avoid re-allocating memory.

Then, the database is updated inside the double loop following

$$\mathbf{D}(i, \mathbf{I}(i, k_b)) = \texttt{NaN}. \tag{12}$$

Setting the elements to `NaN` instead of actually deleting them preserves the structure of the arrays leading to a more efficient management of the memory. Once the double loop is finished, the final steps are to update the number of elements and rebuild the sorted database. The number of elements is results in

$$N_E = N_E - N_R. \tag{13}$$

Finally, the sorted database is rebuilt row by row with

$$\mathbf{S}(i, 1 : N_E) = \mathbf{D}(i, \mathbf{I}(i, 1 : N_E)). \tag{14}$$

Algorithm 2 provides pseudocode that supplements the explanation given in this section.

Algorithm 2 Removing Elements

for $i \in [1, N_D]$ **do**
  for $j \in [1, N_R]$ **do**
    $K_{\text{ind}} = \texttt{ceil}(\mathbf{E}_R(i,j)/m(i) + q(i))$
    $k_b = \mathbf{K}(i, K_{\text{ind}} - 1)$
    {Perform a linear search to find the exact value of k_b}
    while $\mathbf{D}(i, \mathbf{I}(i, k_b)) > E_R(i,j)$ **do**
      k_b = k_b + 1
    end while
    k_b = k_b - 1
    $\mathbf{K}(:, K_{\text{ind}} : N_E) = \mathbf{K}(:, K_{\text{ind}} : N_E) - 1$
    $\mathbf{I}(i, 1 : N_E) = [\mathbf{I}(i, 1 : k_b - 1), \mathbf{I}(i, k_b + 1 : N_E), 0]$
    $\mathbf{D}(i, \mathbf{I}(i, k_b)) = \texttt{NaN}$
  end for
end for
$N_E = N_E - N_R$
$\mathbf{S}(i, 1 : N_E) = \mathbf{D}(i, \mathbf{I}(i, 1 : N_E))$

POTENTIAL FOR PARALLELIZATION

There is potential for the DNDKV to be implemented in parallel. Implementing the DNDKV in parallel would provide the capability to manage a database and search through that database at the same time. This would be equivalent to supporting row-locking as opposed to table-locking, an advantage of the InnoDB over the MyISAM storage engines in MySQL. Database management in parallel could be accomplished by creating two copies of the DNDKV structure. One set of threads/processors would work on adding and removing elements from the first copy of the database (database management), while the other set of threads/processors performed range searches on the second copy. When the first set of threads/processors was done modifying the first copy and the second set of threads/processors was done performing searches, the sets of threads/processors would swap the copy of the DNDKV that they were working on. Figure 1 graphically shows this idea using two different CPUs.

In Fig. 1, CPU 1 handles the search queries while CPU 2 handles the database management. CPU 2 works at maintaining the database until all the desired elements have been added or removed. Once this happens, the switch (center block) switches the copy of the database that CPU 1 and CPU 2 are working on. This allows the database to be updated and search queries to be performed simultaneously.

A DATABASE OF MISSIONS TO CELESTIAL BODIES

Reference [7] describes a newly created database of pre-computed missions to all known asteroids and comets. It includes 10 search dimensions, which are the departure date, arrival date, Δv required to depart from Earth, v_∞ at arrival, phase angle, Earth distance, solar elongation at arrival (Sun-Earth-probe angle), declination of the launch asymptote (DLA), approach angle (formed by the incoming v_∞ vector and the heliocentric velocity of the body), and time of flight. A similar database has been created to demonstrate the utility and power of the DNDKV. This database contains 20 million different missions. Figure 2 shows a histogram of the time it takes the DNDKV to find all

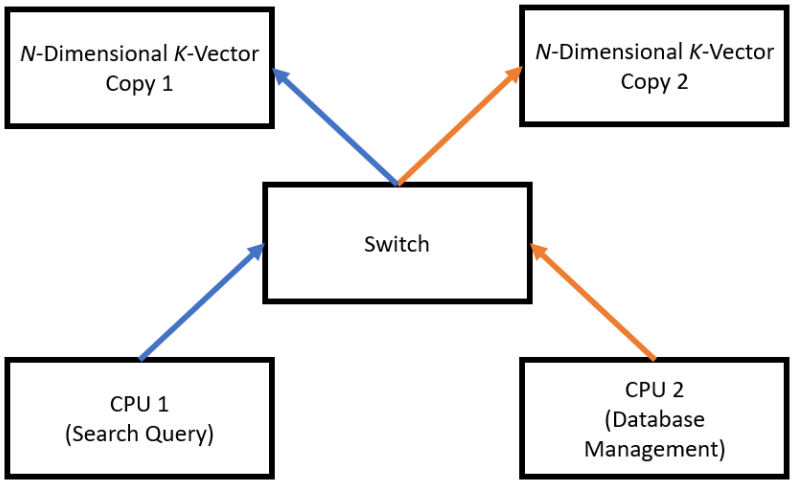


Figure 1: Parallel Database Management System

the elements within a randomly generated search query.

All times reported in the tables and figures in this section were calculated using the `tic` and `toc` functions in MATLAB. These tests were performed on a Windows 10 operating system. The computer used to run the tests had an Intel(R) Core(TM) i7-7700QM CPU running at 3.60GHz and 16.00 GB of RAM.

The average search time was 18.64 seconds and the standard deviation was 3.16 seconds. Figure 2 demonstrates the speed attainable by the NDKV. Moreover, the aforementioned search times were when searching with restrictions on all 10 dimensions. If restrictions are only placed on a few of the dimensions this search time becomes even faster. For example, random search queries on only two of the dimensions had an average search time of 5.73 seconds. Tables 1 and 2 compare the time it takes to rebuild the NDKV with the time it takes to add or remove elements respectively using the DNDKV.

Tables 1 and 2 compare the time it takes to add and remove elements using the DNDKV with the time it takes to completely rebuild the NDKV structure.

Number of Elements Added	Rebuild Time (s)	Add Time (s)
1	117.8	15.98
10	118.7	47.29
100	126.9	381.4

Table 1: Rebuilding the Database Versus Adding Elements to the Database

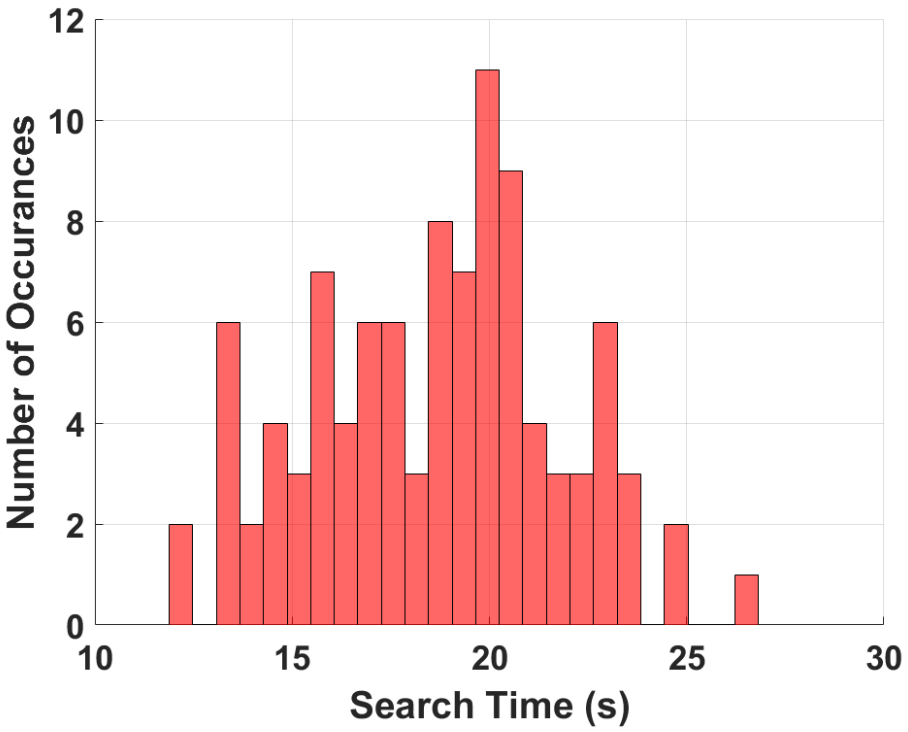


Figure 2: N-Dimensional K-Vector Search Time

Number of Elements Removed	Rebuild Time (s)	Remove Time (s)
1	115.2	14.73
10	114.6	46.54
100	113.9	372.2

Table 2: Rebuilding the Database Versus Removing Elements from the Database

Tables 1 and 2 show that using the DNDKV technique is faster than rebuilding the NDKV from scratch, provided that the number of elements being added or removed is sufficiently low. However, once the number of elements being added or removed becomes large enough, the time to rebuild the database actually takes longer than just rebuilding the NDKV from scratch. The exact number of elements at which rebuilding the NDKV from scratch becomes more computationally efficient than using the DNDKV is highly sensitive to the number of elements already in the database and where in the database the elements are being added or removed. The averages from a Monte Carlo simulation, like the on performed here, can help determine whether to completely rebuild the NDKV or use the DNDKV. However, this type of test will have to be performed for each database if the user wants to find the most computationally efficient solution.

Tables 1 and 2 also show that the DNDKV is most useful when only a few elements are added at a time. For example, if the sample database used here had ten elements added to it every hour, then the DNDKV is a perfect solution, because search queries on the database only have to be stopped for approximately 47 seconds every hour while the database updates.

PERFORMANCE OF THE DNDKV

Most production systems rely on structured query languages (SQL) for managing relational databases. In particular, MySQL is a very popular open-source engine that was developed more than 20 years ago. Therefore, it is important to compare the performance of the DNDKV to standard MySQL storage engines to fully assess the potential of the algorithm. The MySQL server used in the simulations is hosted on a MacBook Pro High Sierra, with an Intel(R) Core(TM) i7 CPU 3.1GHz and 16.00 GB of RAM. For consistency, the DNDKV queries are also run on this machine.

MySQL is written in C and C++ and currently supports several storage engines, each with different features. MyISAM is chosen over InnoDB to run the performance comparisons in this paper as it is often more efficient for reading from the database. MyISAM implements a B-Tree indexing system for fast database lookup, although the query optimizer ultimately decides whether to use indexing or not depending on the estimated cost of the B-Tree search compared to a full table scan.

There are three main operations to interact with a relational database: `SELECT`, `DELETE`, and `INSERT`/`UPDATE`. `SELECT` statements filter the database to find the records that satisfy certain search criteria and operates in read-only mode. `DELETE` and `INSERT` are used to remove and add records to the database, respectively, and `UPDATE` allows to modify specific records that are already present. When writing to the database either in `DELETE` or `INSERT`/`UPDATE` mode, the MySQL engine needs to update the corresponding index table and re-order the data in the table, just like the DNDKV algorithm needs to update the k-vector. Three different tests have been designed to compare the performance of the DNDKV compared to MySQL when operating in `SELECT`, `DELETE`, and `INSERT`/`UPDATE` modes:

- `SELECT` mode: Table 3 lists 10 different queries conceived to produce a diverse set of data intersections. Each query produces an increasing number of matches. For simplicity and without loss of generality, we restrict the queries to the departure and arrival dates, the time of flight (TOF), the departure Δv, the arrival v_∞, and the approach angle γ, defined by the arrival v-infinity vector and the heliocentric velocity of the body:

$$\gamma = \arccos\left(\frac{\mathbf{v}_\infty \cdot \mathbf{v}_{\text{bod}}}{||\mathbf{v}_\infty||\,||\mathbf{v}_{\text{bod}}||}\right).$$

  Figure 3 provides some insight into how the different queries intersect the data under consideration, using queries 2, 3, and 7 as examples.

- `DELETE` mode: all missions to certain asteroids will be deleted. The chosen asteroids are the first one in the database, and those that fall closer to the 25%, 50%, and 75% positions. Removing records in very different positions in the database exercises the data resorting carried out by both the DNDKV and MySQL.

- `UPDATE` mode: all fields except the object ID and the dates will be set to zero for the first record and those that fall closer to the 25%, 50%, and 75% positions. The `UPDATE` mode is chosen over `INSERT` because the latter will always append records at the end of the table, whereas the former forces the storage engine to modify the indexes in the case of MySQL or the k-vector in the case of the DNDKV.

Since the performance of each operation depends strongly on the size of the database, the test database has been split databases of increasing size to assess how each database engine is affected

Id	Matches (%)	Departure date (MJD)	Arrival date (MJD)	TOF (days)	Δv_{dep} (km/s)	$v_{\infty,\text{arr}}$ (km/s)	Appr. ang. (deg)
1	0.02	–	–	$= 300$	< 5	–	–
2	0.78	$(59500, 63500)$	$(63000, 65000)$	$(300, 2500)$	$(1, 6)$	$(5, 15)$	$(50, 150)$
3	3.66	$(62000, 62400)$	$(62500, 64000)$	–	–	–	–
4	4.78	> 64500	< 70000	$(100, 1440)$	< 7	< 10	> 120
5	10.95	–	< 68000	< 1500	–	< 5	–
6	31.96	–	$(65000, 70000)$	–	–	–	$(90, 170)$
7	38.76	–	–	$(365, 1461)$	$(3, 9)$	$(1, 10)$	–
8	71.38	> 60000	–	–	> 6	–	–
9	78.34	–	–	–	–	$(5, 15)$	> 100
10	86.41	–	–	< 1461	–	–	–

Table 3: Definition of the test queries

by this factor. Table 4 describes each test database, which has been generated by randomly sampling asteroids from the master database in [7] until the database reached the requested size.

Database Id	Records	Data Size (MB)	Index Size (MB)	Total Size (MB)
DB-A	20,000	1.01	2.30	3.31
DB-B	63,246	3.20	7.23	10.43
DB-C	200,000	10.11	22.78	32.89
DB-D	632,456	31.97	72.05	104.02
DB-E	2,000,000	101.09	227.88	328.97

Table 4: Description of the test databases

The random sampling from the master database ensures that the data in each database is equally distributed, as proven in Fig. 4.

`SELECT` Statements

The performance of the NDKV when filtering data is compared to MySQL by running the queries in Table 3 on all the databases in Table 4. For each database size, the relative of speedup of NDKV with respect to MySQL is presented in Fig. 5. Depending on the query parameters, the NDKV show speedups of more than a factor 6. The intersection process when using the NDKV is more sensitive to the size of the database than MySQL. This apparent performance loss can potentially be alleviated, at least partially, when working with a C/C++ version of the NDKV that accelerates the intersection step.

Query 10 exhibits the best relative performance and it corresponds to a search with constraints on only one parameter. Intuitively, searching along one unique dimension is extremely easy for the NDKV because the underlying data sorting. As more variables are to be intersected, the computational cost will increase. Queries 3 and 5 follow in terms of performance improvements with two and three intersected variables, respectively.

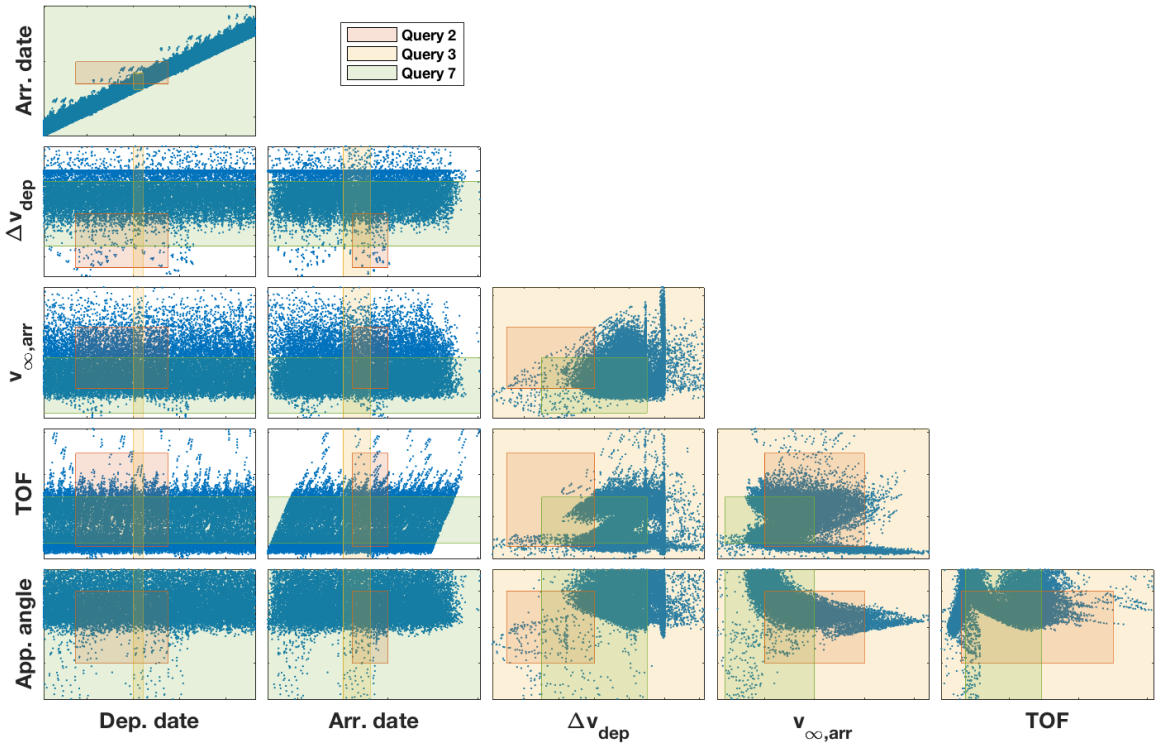


Figure 3: Visualization of queries 2, 3, and 7 intersecting the data

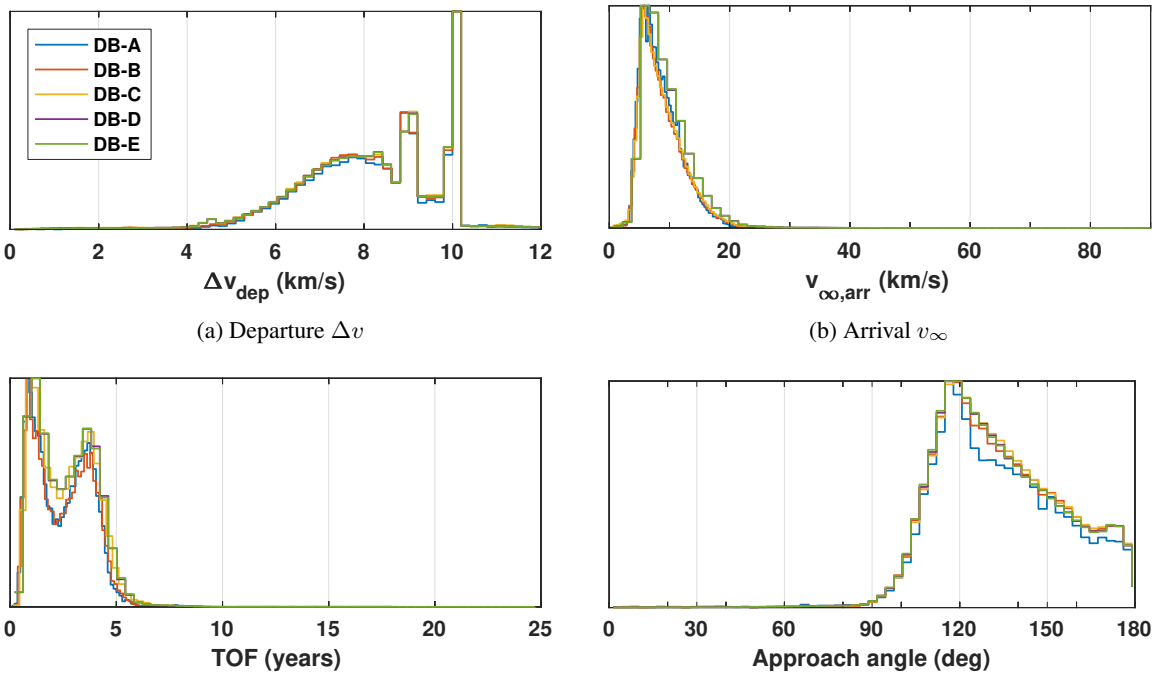
The case of Query 1 is interesting because it shows how MySQL is more efficient when resolving equality constraints. In particular, the index on the time-of-flight variable allows MySQL to access the table by reference rather than by scanning, reducing the estimated cost by a factor of 45. This is a good example of the improvements in the performance that the optimizer can potentially introduce when choosing the right use of the indexes. There are, however, some situations in which conservative cost estimates make the optimizer decides to use an index while a direct scan of the table would be more efficient. This phenomenon explains the relatively poor performance of MySQL when resolving Query 4 for the first and last database sizes.

`DELETE` Statements

Figure 6 compares the performance of the `DELETE` calls. MySQL outperforms the current implementation of the NDKV because of the intrinsic cost of updating the k-vector. The latter algorithm is again more sensitive to the size of the database, resulting in performance drops as the number of elements in the table increases.

`UPDATE` Statements

Being both "write" operations, the relative performance of the `UPDATE` calls is similar to the `DELETE` calls. Figure 7 shows that MySQL might be several orders of magnitude faster than the NDKV as the database size increases.



(a) Departure Δv

(b) Arrival v_∞

(c) Time of flight

(d) Approach angle γ

Figure 4: Relative distribution of records in the test databases

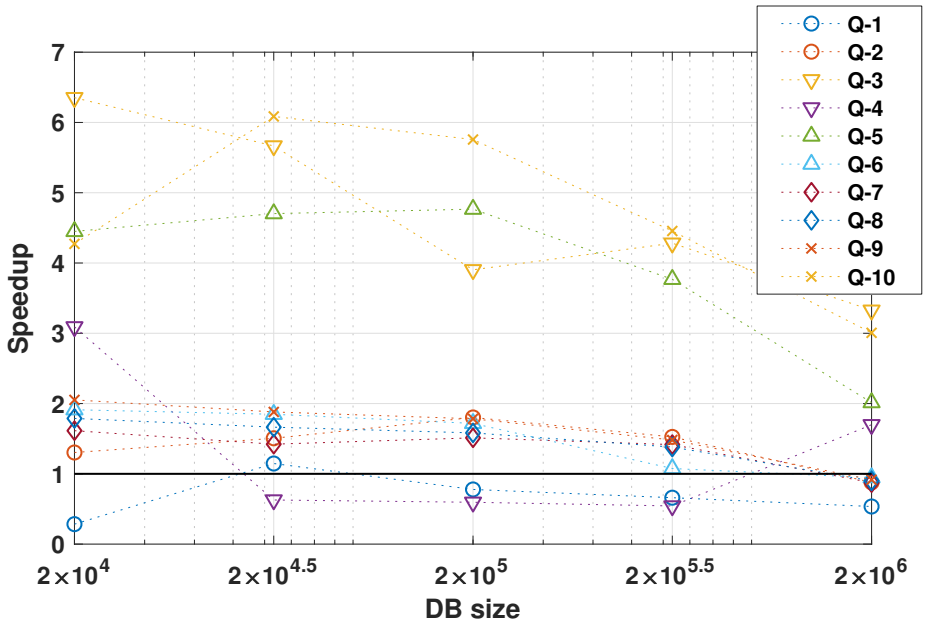


Figure 5: Speedup of the NDKV compared to MySQL's `SELECT` mode

SUMMARY AND FUTURE WORK

In summation, the dynamic n-dimensional k-vector (DNDKV) is a powerful technique that extends the NDKV to databases that have elements added or removed over time. This increases sig-

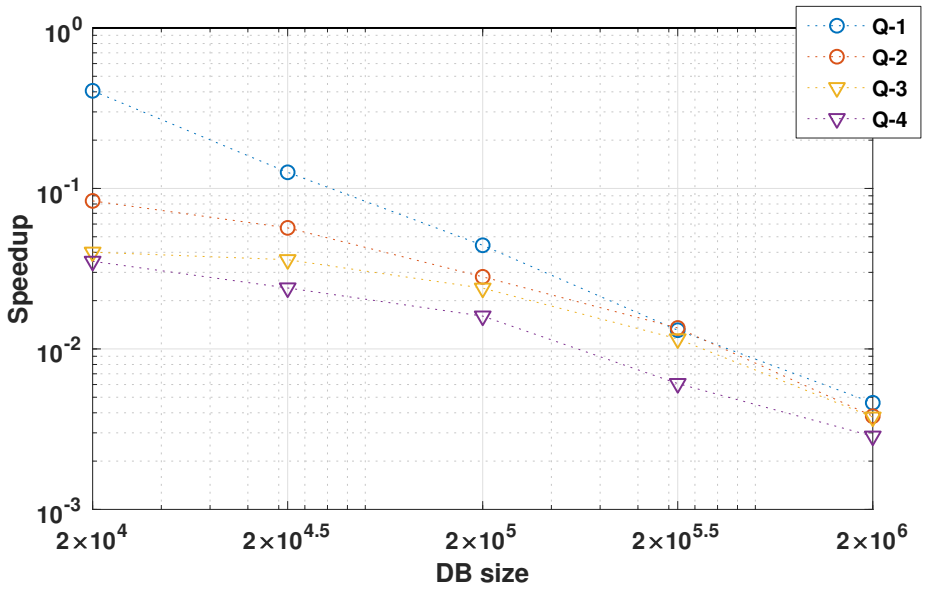


Figure 6: Speedup of the NDKV compared to MySQL's `DELETE` mode

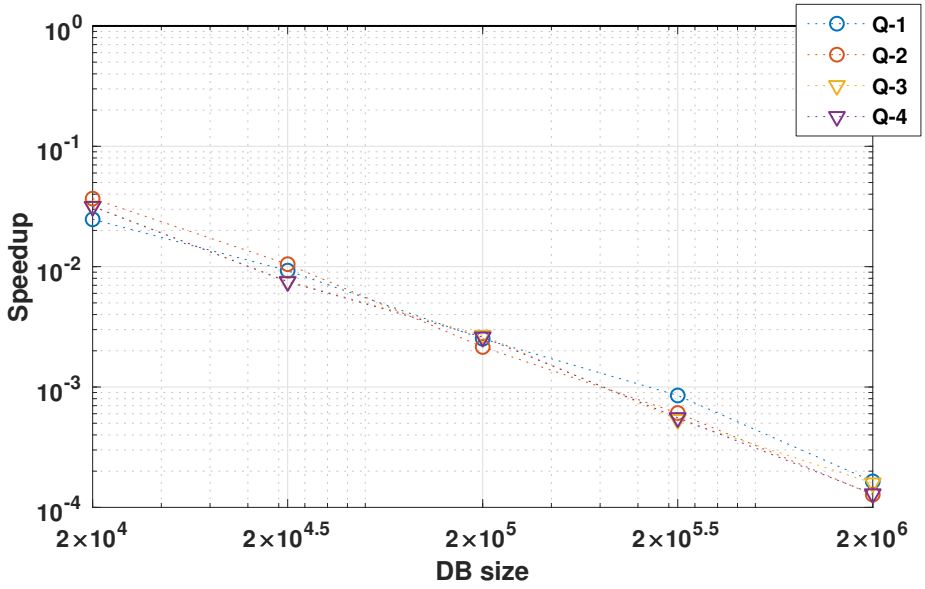


Figure 7: Speedup of the NDKV compared to MySQL's `UPDATE` mode

nificantly the range of applications of the NDKV as it is now an appealing alternative for managing dynamic databases. The DNDKV is most useful when the updates to the database are frequent, but only have a few elements being added or removed. Moreover, the elements being added to the database must fall within the current range of the k-vector tables.

Performance tests suggest that important improvements in the performance of search queries can be expected when comparing the NDKV with state-of-the-art production engines like MySQL. Statements that modify the database are still more costly but, for read-intense applications, overall performance gains should be expected.

Future work should investigate running the DNDKV in parallel with database search queries. Doing so would allow one to continually search through the database with no interruptions for database updates, similar to the row-locking capabilities of the InnoDB engine. Furthermore, future work could include finding a better method than Monte Carlo simulation to predict how many elements need to be added or removed before completely rebuilding the NDKV structure becomes more computationally efficient than using the DNDKV. Implementation improvements might reduce the cost of "write" operations associated to re-computing the k-vector.

The DNDKV loads the database in RAM memory, which explains in part the speedups when reading the data. This behavior can limit the size of the databases that can be efficiently handled by the DNDKV. Splitting techniques should be investigated in order to increase the potential of the method.

ACKNOWLEDGMENTS

Part of this research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

REFERENCES

[1] D. Mortari, M. A. Samaan, C. Bruccoleri, J. L. Junkins, The *Pyramid* Star Identification Technique, ION Navigation 51 (3) (2004) 171–183.

[2] Star identification has never been so fast and accurate, `http://www.esa.int/gsp/ACT/news/archive/84_oct_2017_star_trackers.html`, accessed: 03/18/2019.

[3] D. Mortari, J. Rogers, A k-vector Approach to Sampling, Interpolation, and Approximation, AAS *The Journal of the Astronautical Sciences* 60 (3) (2015) 686–706.

[4] D. Mortari, Memory Adaptive k-vector, in: 2014 AAS/AIAA Space Flight Mechanics Meeting Conference, Santa Fe, NM, 2014, AAS 14-207.

[5] D. Mortari, B. Neta, k-vector Range Searching Techniques, in: Advances in the Astronautical Sciences, Vol. 105, Pt. I, 2000, pp. 449–464.

[6] D. Mortari, C. Leake, S. Borissov, The n-dimensional k-vector with applications, in: 2018 AAS/AIAA Space Flight Mechanics Meeting Conference, Kissimmee, FL, 2018.

[7] J. Roa, A. B. Chamberlin, R. S. Park, A. E. Petropoulos, P. W. Chodas, D. Landau, D. Farnocchia, Automatic design of missions to small bodies, in: 2018 Space Flight Mechanics Meeting, 2018, p. 0200. doi:10.2514/6.2018-0200.