Introducing SimiLite
Enabling Similarity Retrieval in SQL

Kristian Veøy

Master of Science in Computer Science
Submission date: June 2011
Supervisor: Magnus Lie Hetland, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

This project has implemented SimiLite, a plug-in to SQLite which enables the usage of metric indices in SQL tables. SimiLite can easily be extended with different indices, and the indices LAESA and SSSTree has been implemented and verified.

This project has also implemented a framework for easy comparison of the indices within SimiLite.

It was found that while SimiLite causes a slow-down of about 5-10 compared to the reference solution for a light metric, this will balance out quickly once the cost of the metric increases.

Contents

Chapter 1

Introduction

In later years the amount of data such as images, video and audio recordings has flourished, and there are not many good means for searching for content based on anything but a textual description. Similarity retrieval represents an interesting approach to providing new possibilities for searching in such complex data.

Similarity retrieval works by retrieving elements that are similar according to some measure. This could for instance be used to locating images similar to one you have taken, or find the video a still image is taken from. Other opportunities lies within scientific research, where for instance locating similar sequences in the genome is of interest. The notion of similarity is very broad and there are doubtlessly many more possible applications that are not yet known.

Similarity retrieval has proved to be an expensive undertaking, however. While numbers and words have a total ordering, the objects used in similarity retrieval often has little indexable structure, and therefore other methods must be used to efficiently perform the query. Additionally the action of calculating the similarity between two objects is itself usually quite expensive, so even if a simple scan of the data would otherwise be enough, it might become too expensive when taking this cost into account. Because of this special types of indices has been designed.

One type of such indices for use with similarity search is called metric indices. Metric indexing allows more efficient similarity retrieval by setting some requirements on the similarity measure. It has had little wide usage, and one reason might be because is not readily available. By integrating a metric index into a standard SQL database engine, the threshold for playing with the technology will decrease.

This project is a continuation of an effort to implement a metric index in SQLite's virtual table functionality. Previously a proof-of-concept was completed, but testing and analysis was incomplete. kNN-queries (k Nearest Neighbours), which are easier for users to relate to, was also completely absent. Since it is virtually useless without knowing the performance, this project aims at fixing this issue, and at the same time look at what improvements can be done with the implementation. The report is meant to be accessible to people with no prior knowledge of metric indices or spaces, but familiarity with these subjects should make it easier to understand.

One thing that must be known about metric indexing is that it will in general not offer anything but a constant factor speedup. An asymptotically better solution is not generally possible. The constant factor of those improvements, however, can be significant, and in some cases a good enough speedup can turn a previously impossible feature into reality. For instance an online search operation being performed in 300 ms instead of 3 seconds. Studies have shown that a latency of over 3 seconds can cause users to look for other alternatives[1], which means that such a slow feature is probably unacceptable. By speeding up the similarity query by a possible factor of 10, new possibilities are opened. Another possibility could be a data mining operation ordinarily taking 5 days, which in reality means a latency of a week, to being done overnight (12 hours). This could make it much easier to analyze results, and would also only be a speedup of 10. A practical example; in [2] a speedup of 300 was achieved based on a Ptolemaic[3] distance index while the metric index version had a speedup of 170. The similarity measure for this paper was the signature quadratic form distance, a very expensive metric possible to use in comparing multimedia data.

This report has two general goals; being a quick introduction to metric indexing for people who are new to the subject, and to convince those already involved that SimiLite can provide a good way to develop metric- and other distance-based indices. It is also the hope that this report can act as a guide in the event that this project is continued.

1.1 Similar Efforts

The similar efforts section is the same as for the preparatory project, and is included here for completeness.

Spatial indices, which are available for many popular relational database engines([4, 5, 6, 7, 8]), can provide efficient searching when the data are coordinates. For other types of data, or when the number of dimensions is high, it is not usable. If, for instance, one were to index images, chances are a new solution would have to be developed from scratch, or something existing would need to be modified. This is a steep curve to climb if it is only wanted to assess the potential value such an index could add to a project.

Yao et al.[9] looks at finding the *k* Nearest Neighbours (kNN) in existing relational databases using only SQL operators which the query optimizer can the optimize further. This could perhaps be used to implement an index within this plug-in which access existing tables instead of creating new ones, and giving it a simpler interface.

Kumaran et al.[10] used the Generalized Search Tree (GiST) feature in PostgreSQL to implement an M-tree to speed up their implementation of a crosslingual database query engine.

> *The GiST provides a possibility to create custom data types with indexed access methods and extensible set of queries for specific domain experts not a database one.* [11]

While the project itself is not relevant, the fact that they needed a metric index and chose to implement it behind an SQL-interface is. Additionally the preferred index was not supported by GiST and therefore not used. Had the SQLite plug-in been available (and they used SQLite instead of PostgreSQL) it could have fulfilled that role, and the preferred index could have been implemented.

Bagge[12] writes about how one can utilize standard database mechanisms to enable similarity search. This is essentially what is done in this project, only his implementation is in java, and is accessed via an API instead of through SQL. He also looks at different methods which can be used in the underlying database engine to speed up the index, some of which could be utilized in the plug-in.

Bioinformatics is one field in which researches the use of metric indices, and Molecular Biological Information System (MoBIoS)[13] has been developed to provide metric indices in this situation. MoBIoS is implemented in java with the Mckoi open source relational database engine and provides SQL-like syntax through an extension to SQL called M-SQL. MoBIoS uses MVPT as the only metric index and therefore doesn't allow comparisons of different indices for different purposes. MoBIoS is designed with bioinformatics in mind, but also allows user-created metrics to be specified. MoBIos is very similar to this project, but seems to be more focused on its bioinformatics use-case.

1.2 Report structure

The remainder of this report has been partitioned as follows: First some necessary background material on metric indices and SQLite is presented in chapter 2. Thereafter an overview of the design of SimiLite is outlined in chapter 3, and the implementation is summarized in chapter 4. An attempt at estimating the impact of using SQLite is done in chapter 6, and the results of testing different implementations is shown in chapter 5. Chapter 7 finalizes the report with some concluding remarks and notes for possibilities of future work.

Chapter 2

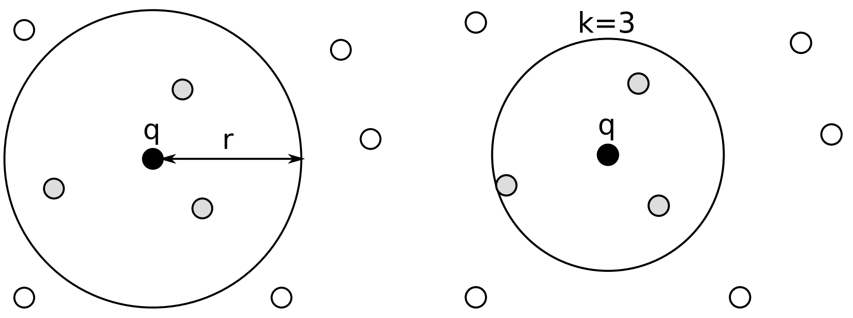
Background

Knowledge of databases, SQL and indices in general is assumed. An example reference for this material is Ramakrishnan and Gehrke(2002)[14]. A familiarity with algorithms, data structures and algorithm analysis is also expected. An example reference for this material is [15]. Knowledge of the C programming language and the C preprocessor are an advantage, but should not be necessary.

2.1 Similarity Retrieval

Similarity retrieval means fetching objects based on their similarity to a another object. The similarity measure, hereafter referred to as distance (actually the closely related dissimilarity measure), is represented through a number and will vary according to the type of objects and the features that are deemed interesting. Similarity retrieval queries often falls into one of the following categories or variants thereof:

- Find objects whose distance to some specific query object is within a given range or radius r. A range query (see figure 2.1a).
- Find the objects whose distance to some specific query object is closer than for all other objects. The number of such objects is usually denoted as k and the query is referred to as a k Nearest Neighbour query, or a kNN query (see figure 2.1b).
- Find pairs of objects which are similar in some fashion. Note the lack of a query object. This case is not covered in this report.



(a) Range query (b) kNN query

Figure 2.1: Query types.

An example of similarity retrieval could be to locate places restaurants in the vicinity. The "similarity" could include features such as type of food and prices, where the goal is to find a restaurant (or k) fitting to your needs (for instance close, not too cheap and Italian). This is an extremely simple example and realistic similarity queries could include many more features.

Without knowing anything more than the distance measure, the only way of performing these queries in general is comparing the query with the possible objects one by one. This is referred to as a linear scan, and is prohibitively expensive when the cost of calculating the distance is high. More structure is needed to avoid the linear scan, and one possibility would be using a metric index. Possibilities for other types of distance based indices exist, and they can be implemented in SimiLite, but metric indices seemed like the standard and is therefore received the primary focus.

2.2 Metric spaces

Before explaining what a metric space is, an explanation of what a space is is in order. A simple explanation would be that a space is a set X of possible objects and an "instance" of the space is some $X' \in X$. This instance would be the contents of the table in the database for our purposes, with the number of objects as $n = |X'|$.

This simple space has no structure to build on, and is therefore difficult to index. A metric space adds some indexable structure by requiring that a distance measure d exists between every pair of objects p, q, and that the following holds for d:

1. $d(p, q) \geq 0$. Intuitively distance is always positive.
2. $d(p, q) = 0 \equiv p = q$. This means that an object can only have 0 distance to itself.
3. $d(p, q) = d(q, p)$. The distance to and from is the same, independent of ordering.
4. $d(p, o) + d(o, q) \geq d(p, q)$. The triangle inequality. The intuition is that there are no magic shortcuts; going via some third location cannot yield a shorter total distance.

The last requirement is what provides structure and makes it possible to estimate the distance without calculating it. Such a function d is called a metric, thereof the name metric space.

The simplest distance metric is perhaps the 0-1 metric where $d(p, q)$ is 0 if $p = q$ and 1 otherwise. While valid, it is not very useful. A much more useful metric for the sake of understanding, is that of shortest travelling time between locations; for instance shortest possible flight time between cities. Naturally it is not possible to spend a negative amount of time flying, and, unless you are not actually travelling (Oslo to Oslo for instance), some time will be spent. It takes the same amount of time to fly back again (when disregarding winds and jet streams and the like), and if it had been possible to spend less time flying from Oslo to Trondheim by landing in Bergen than by going directly, then that would effectively become the "distance".

Some additional examples of spaces with examples of metrics (there can be many for a given space) are:

Universe	**metric**
Numbers	difference ($d(q, p) = \|q - p\|$)
Cartesian coordinate system	Euclidean distance ($d(p, q) = \sqrt{\sum_i (p_i - q_i)^2}$)
Words	Levenshtein/edit distance; the number of letters to be added/removed/changed to make one word into an other
Documents	With the words as dimensions the angle can be measured

Many of the metric spaces have a large number of dimensions. The objects to be indexed from this space, however, might not be as spread and here the concept of intrinsic dimensionality comes in. A simple example in the 3-dimensional euclidean space would be if all the objects lie on a single line. There are 3 dimensions, but the intrinsic

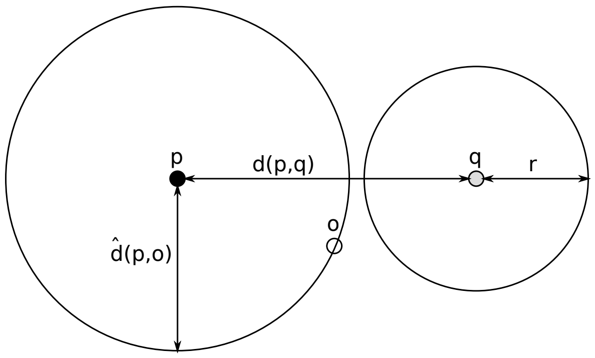


Figure 2.2: Automatic exclusion of the object o. Note that it could have been anywhere on the circle and that $\hat{d}$ means an upper bound.

dimensionality would be 1.

2.3 Metric indices

A metric index is an index which utilize the triangular inequality of a metric to enable more efficient queries in metric spaces. In the situations where a metric index is preferable, the cost of calculating the distance is usually prohibitively high, and therefore often the sole goal, or at least much higher prioritized than other goals is that of minimizing the number of such calculations. The cost of constructing the index is also largely disregarded, as this cost will be "amortized" over the queries run against it.

Central in many indices is the concept of a pivot, which is simply an object in the dataset for which pre-calculated distances to some subset of the other objects in the index exist. For the typical pivot the distance to all the objects in the index has been stored. Also central is the concepts of automatic inclusion and exclusion of objects. It is a match if an upper bound on the distance is within the range, and it is not if the lower bound is outside. Since there usually are more objects outside than inside the range, automatic exclusion is more useful. Automatic inclusion is extra useless in this context as users of the table would also often like to know the distance, in which case the inclusion was redundant as the distance had to be computed anyway.

One version of automatic exclusion which is used often is illustrated in figure 2.2. If an upper bound, $\hat{d}$, for the distance between p and o is known beforehand (d̂, and the distance between p and q is calculated the object o can be automatically excluded if $d(p,q) - \hat{d}(p,o) > r$.

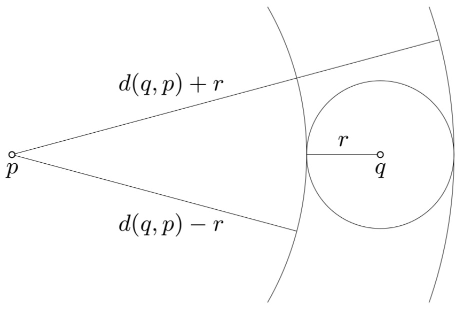


Figure 2.3: Depiction of how the pivot filtering works. Any object o within the inner circle, as seen from p, fulfills $d(p,o) < d(p,q) - r$. Had it also been a valid match ($d(o,q) \leq r$), it could have been used as a shortcut between p and q: $d(p,q) \leq d(p,o) + d(o,q) < d(p,q) - r + r < d(p,q)$. An equivalent statement can be similarly made for the objects outside the outer circle, $d(p,o) > d(p,q) + r$, so all of these can be automatically excluded. (Figure taken from [16] with permission.)

Proof:

$$\begin{aligned} &d(q,o) + d(o,p) \geq d(q,p) \\ &\qquad d(q,o) \geq d(q,p) - d(o,p) \geq d(q,p) - \hat{d}(o,p) > r \\ &\qquad d(q,o) > r \end{aligned}$$

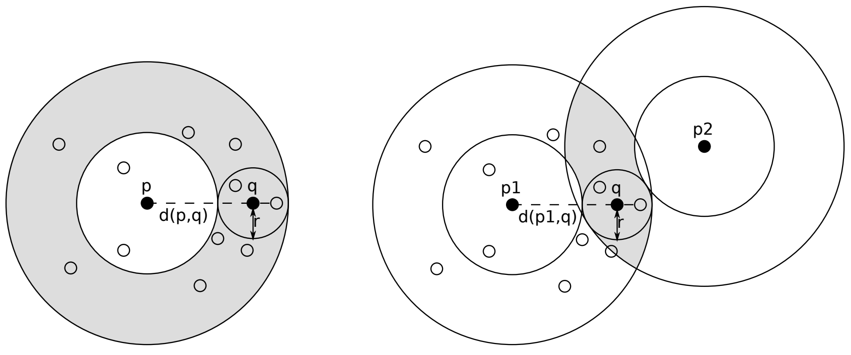
When drawn as a circle, as in figure 2.2, the distances seem very straightforward. This is really only valid in the 2-dimensional Euclidean space, however. In other spaces the range might not be a circle, it would for instance be a square for the metric $d(p,q) = \sum_i |p_i - q_i|$. It could even impossible to draw in an easily understandable way (documents comparison for instance).

For the remainder of this report p, q and o will symbolize respectively a pivot object, the query object or some arbitrary object.

For a thorough introduction to metric indexing and the related concepts see [16, 17, 18].

2.3.1 LAESA

LAESA[19] is a pivot-based metric indexing scheme with roots in the earlier Approximating and Eliminating Search Algorithm(AESA)[20, 21]. AESA simply lets every object be a pivot and stores the distances between all pairs in a 2-dimensional array. The construction



(a) 1 pivot. (b) 2 pivots.

Figure 2.4: Illustration of the area containing possible matches when filtering with pivots.

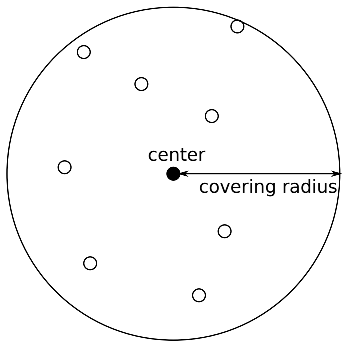


Figure 2.5: The structure of an SSSTree cluster.

phase thus requires n^2 distance computations, which, while in the construction phase, is still quite high. LAESA attempts to improve on this by using a constant number, p, of the objects as pivots, and using these to automatically exclude objects during queries based on their distance to the pivots. As p is constant this is a linear time algorithm: $O(n \cdot p) = O(n)$. In general $n \gg p$.

The basic premise of the filtering is shown and explained in figure 2.3, and a visualization of how the search space can be constricted is shown in figures 2.4a and 2.4b for respectively one and two pivots. There are several ways to select the pivots used by LAESA, but in general it is attempted to choose pivots that are far away from each other, as this has shown to give the most filtering power.

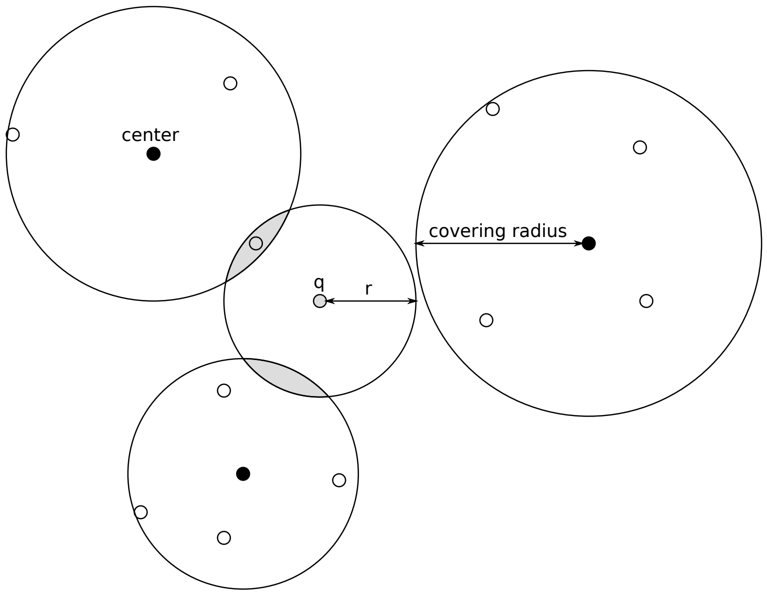


Figure 2.6: Shows overlap between the query object and three clusters. The areas of possible overlap are greyed out. Note that it is possible for there to be no objects in the possible overlap zone.

2.3.2 SSSTree

SSSTree[22] is a tree based structure where each parent has 2 or more children. The nodes, or clusters in SSSTree, consist of a center object, the children and the distance to the most distant object below it. This distance is called the covering radius (see figure 2.5). At the top level there are only a collection of such clusters.

When performing a range query, the top level clusters will be put in a list, and the query object will be compared to each of the center objects in turn. If there is overlap (see figure 2.6) the children of the cluster is added to the back of the list, otherwise they are ignored. If the center object itself is a match it will be treated as such. The process continues until there are no more overlapping clusters, or a leaf cluster is reached. The leaf cluster, or bucket, contains objects which are not center objects, and each of these will be tested in turn for a match.

The construction phase is the key to SSSTree. Here a process called Sparse Spatial Selection (thereof the name SSSTree) is used to choose centers. This is done by letting objects become new centers if they are a certain distance away from the other current centers (see figure 2.7 for an illustration of this process). This boundary distance is chosen

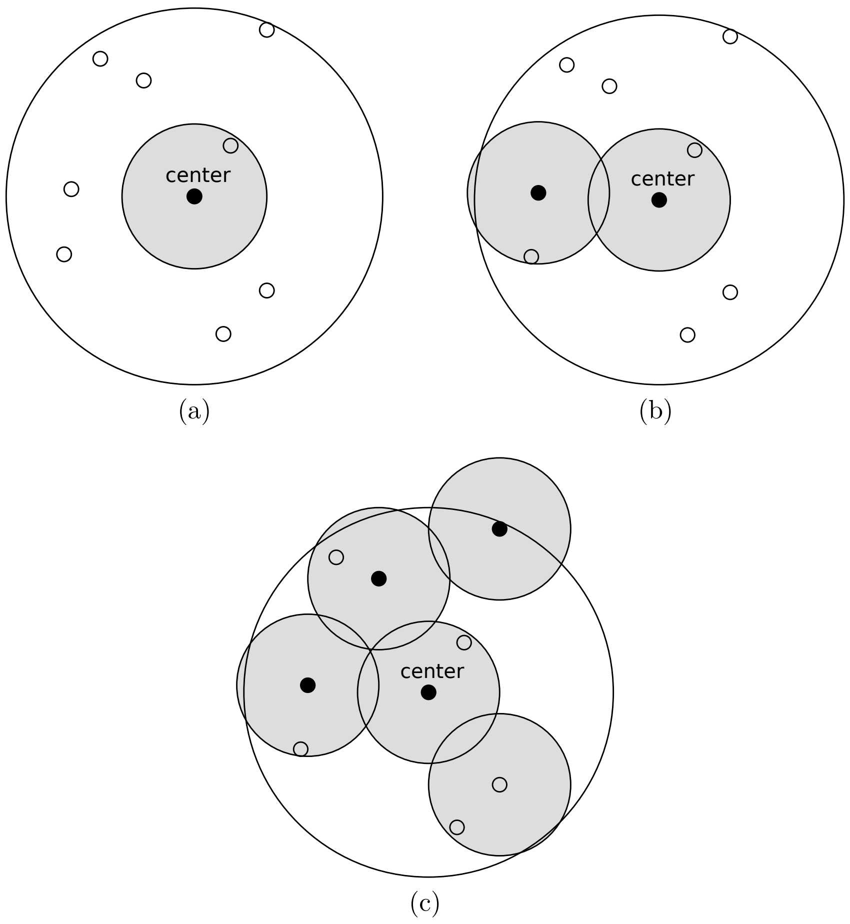


Figure 2.7: The process of selecting new centers in SSSTree. The black circles represent the new clusters, and the grey areas where it is too close to a center to become a new one.

as $M \cdot \alpha$, where M is the maximum distance between any two objects in the cluster. As was shown in [22], this distance can be estimated as 2·covering radius. The constant α was empirically shown in the paper to be optimal between 0.35 and 0.4. The nodes which does not become new clusters are inserted into the nearest cluster. A bucket is split into clusters only if the number of elements are above a certain threshold.

This process of choosing clusters is very dynamic, and enables SSSTree to adapt its structure to the distribution of the objects in it, ideally matching their intrinsic dimensionality.

2.3.3 SISAP Metric Space Library

SISAP[23] is a conference for the topic of similarity search, and they have given out a library for metric indexing called the Metric Space Library, hereafter referred to as MSL. MSL implements various metric indices for three different kinds of spaces: d-dimensional vectors, strings and documents. MSL also contains example datasets for the different spaces, and generators for random datasets in some instances.

Among the indices are pivots, mvp(Multi Vantage Point tree) and bkt(Burkhard Keller Tree) which are used for comparisons with SimiLite. Pivots is a version of LAESA where the pivots are chosen as the first p objects of the dataset.

NASA

The example space from MSL; a set of 20-dimensional Euclidean feature vectors based on 40150 images downloaded from NASA. This is considered a relatively small dataset and the metric is also cheap, therefore this set mainly tests overhead of solutions. It is also not necessary to test SimiLite on too many different and more difficult spaces, as that would test the indices themselves more than the implementation.

The entire dataset is inserted into SimiLite in the tests. This means every query object will already exist in the database, a fact which potentially could be exploited by an index. However, none of the indices implemented in SimiLite does this, and this allows us to use as many elements as possible and know that there will always be at least one match.

For range-queries two radii were used: 0.2 and 0.9. A radius of 0.2 returns around 25 elements, but with much variance. This is called a small query. A radius of 0.9, a large query, returns about 10% of the elements and is usually the point at which a linear scan could be considered equivalent. This is therefore where performance for the indices should be worst, and can give perspective on what to expect from SimiLite.

2.4 SQLite

SQLite[24] is a very lightweight, open source library that implements an SQL database engine (written in C). Contrary to most DBMSs, SQLite is not run in a different process which must be connected to, but instead is run in-process. For most purposes SQLite will be perfectly adequate to replace alternative DBMSs, and can in many cases be more efficient because it does not need to set up the connection. The downside to using SQLite is the fact that it has not been designed with many concurrent and/or complex queries in mind, and if used in these scenarios it will be inferior.

An SQLite philosophy is that anything can be put into a column, regardless of table definition. This means that for example inserting text into a column declared integer is acceptable. The table definition should rather be used by programmers as a guide to what the column is intended to contain. The tables in SQLite will always have a unique integer key associated with each row. This key, the *rowid*, is indexed with a B-tree (the only index supported natively by SQLite), which means that most operations will at least be $O(\lg n)$. If the table is declared with one column as "INTEGER PRIMARY KEY", this will be bound to the *rowid*, and the column will be restricted to integers. The index on *rowid* is also the actual table.

If a table definition has a UNIQUE constraint on some columns, an index will be constructed on the fields. This index will also be used automatically as an ordinary index, a fact which is used in this project as a simple way of setting up the index (no need to name it explicitly). By including the id as the last field it is ensured that the entry in fact is unique. If the fields in the index contain enough data to satisfy a query then the main index will never be visited, and so attaching a few extra columns can increase performance without increasing memory usage much.

The use of SQLite was largely because of its Virtual Table functionality, an equivalent of which has not been found elsewhere. For a complete introduction of SQLite see Owens(2006)[25].

2.4.1 SQLite Virtual Table

SQLite has support for something it calls a Virtual Table, which is a way to masquerade functionality typically not found in SQL as an ordinary table. Two examples of virtual tables available from SQLite itself is an R*-tree[26] module and a full text search facility for document collections called FTS3[27]. There is also a module called SpatiaLite[8] which allows spatial queries; this was the original inspiration for SimiLite.

The Virtual Table in SQLite is just a collection of methods to call for creating tables, inserting elements and so on. SQLite translate SQL queries into calls to these methods. Internally they are represented in a struct of type *sqlite3_module*, and this must be loaded dynamically during runtime. This is done by calling the two function *sqlite3_enable_load_extension* and *sqlite3_load_extension* with the name of the library-file to load. SQLite calls a function called *sqlite3_extension_initfile* in the library(which must exist), and in this method the struct containing pointers to the methods describing the virtual table is returned.

The relevant methods one needs to implement are:

xCreate/xConnect Used to create a new virtual table and/or set up the state required to interact with an existing one.

xDestroy/xDisconnect Free up state from above and/or remove an existing virtual table.

xBestIndex Used by SQLite to find a good query-plan by returning a cost estimate for a given query. SQLite will call this method with different parameters, and based on the output choose the best plan.

xOpen/xClose Sets up or frees the state required of a pointer into the table. This is used when stepping through a query.

xFilter Positions a pointer based on output from the call to *xBestIndex* for the best plan.

xNext Moves to the next matching row.

xColumn Fetches the values of the columns of the current row.

xUpdate Used to insert, delete or modify rows.

The naming scheme with the prepended x is an SQLite naming convention which was kept for this project.

The syntax for setting up a virtual table is:

```sql
CREATE VIRTUAL TABLE VirtualTable USING VTModule(arg1, arg2, ...);
```

Here "VirtualTableModule" is the name of the virtual table module loaded into SQLite. The arguments are passed on to the virtual table module's xCreate function.

The virtual table functionality provides no other API into the SQLite engine than the standard for ordinary SQLite usage. This means that for the virtual table to have any persistent storage it needs to create this by itself, for instance as a separate file. A better solution would be using ordinary tables for this purpose. The tables created by the virtual table will hereafter referred to as shadow tables, a name used in the documentation of R*-trees. From this also comes the naming convention adopted by this project to avoid name-clashes; shadow tables are named as *_*description* where the * is replaced by the name of the virtual table created, and description should describe the purpose of the table. For instance in R*-Tree the following tables are created for a virtual table named "rtree": rtree_node, rtree_parent and rtree_rowid. The shadow tables can be queried the same as all other tables, but modifying the contents will of course risk corrupting the virtual table. The shadow tables are created in *xCreate* and removed in *xDestroy*.

The input to *xBestIndex* is a struct containing constraints as column-id and one of the operators =, <=, <, >=, > and MATCH. This means that as the virtual table exist today, it is not possible to gain informations on any function-constraints (like distance(q, o) $<$ 1.0), this constrains how the syntax for similarity queries can be made. *xBestIndex* also receives information about the output order, and may avoid having SQLite doing a separate ordering pass it these can be fulfilled easier internally. It is also possible to have SQLite check the constraints on the query instead of doing it explicitly, this is in fact the default behaviour.

Chapter 3

Design

3.1 SimiLite

SimiLite does not work as a regular index in that you add an attribute to a column, instead it is a table of its own. This table will require a column for object at the very least. In addition, since SQLite constructs it anyway, an id field is also added. This is also how it is intended for additional information to be connected to the object, for example:

```
CREATE TABLE extra (
        id INTEGER PRIMARY KEY,
        FOREIGN KEY reference_id REFERENCES metric_table(id),
        date_added DATE
        );
```

The *_main table itself looks like:

```
CREATE TABLE *_main (
        id INTEGER PRIMARY KEY,
        object BLOB NOT NULL
        );
```

The design of the syntax for SimiLite begins with a discussion of what the queries ought to look like, and what is currently possible with the virtual table module. After the syntax for range- and kNN-queries has been established the construction and insertion syntax will follow.

3.1.1 Range query

A query like the following will seem very attractive at first:

```
SELECT * FROM t WHERE distance(object, query) < 1.0;
```

This is not currently possible, however, as such a function constraint would be checked by SQLite on return of every row, and SimiLite would never get access to the actual radius. Since the indices utilizes this range to minimize the number of metric computations, it must be accessible. The only possibility through which the variable can be accesses is a constraint on a virtual table column, and it must be $<$, $\leq$, $>$, $\geq$, $=$ or MATCH as mentioned in background. Some key examples are:

```
SELECT * FROM t WHERE column < 1.0;
SELECT * FROM t WHERE column == 'string';
SELECT * FROM t WHERE column MATCH '1.0';
```

Note that MATCH is a function, but the only which can be used like this.

The MATCH function is a possibility for the range-query syntax, but since the value must be accessible it would end up like:

```
SELECT distance(query, object) FROM t WHERE distance MATCH '<:1.0';
```

This is not very intuitive. Instead the following mental model was adopted: There are two modes of accessing the virtual table; with and without the query object. Without the query object the virtual table is a simple table where only the id and object is accessible. With the query object as an argument it can, together with object, act as an index into a distance matrix (see table 3.1). Then it is more natural for the distance and query-object to be available in addition to id and object. This also makes it more intuitive to constrain the distance for the range queries:

```
SELECT id, distance FROM t WHERE query = _ AND distance < 1.0;
```

The virtual table functionality supports this view by supporting the use of virtual columns. These columns will not be chosen as default during a "SELECT * FROM" query, which is good considering the query would fail otherwise when the query object is not set.

id	object	distance			
		query_1	query_2	query_3	query_4
1	obj_1	0.05	0.49	0.64	0.83
2	obj_2	0.66	0.02	0.10	0.91
3	obj_3	0.58	0.98	0.80	0.70
4	obj_4	0.05	0.69	0.84	0.55
5	obj_5	0.75	0.96	0.97	0.73
6	obj_6	0.08	0.48	0.62	0.34

Table 3.1: One way to visualize lookups in SimiLite.

3.1.2 kNN query

For the kNN query it would have been possible to utilize the LIMIT clause as the k, but it is at the time of writing not possible to access this from a virtual table. In addition it could return incorrect results in some cases:

```
SELECT * FROM table WHERE id < 1000 AND query = _ LIMIT 5;
```

A different example is joins with other tables where there might be constraints on the other tables. Since SimiLite could not know beforehand which of the returned elements would be filtered away, it would have to perform the kNN query with a larger value, but this would be unknown and therefore the search would have to be more dynamic than standard kNN-queries. While it would be nice to support such queries, it would be outside the scope of this project.

In the end, since no other good way was found for supporting kNN-queries, a temporary solution was chosen:

```
SELECT * FROM table WHERE query = _ AND object MATCH 'kNN:5';
```

The MATCH function is not very descriptive in this case, but it is, as mentioned earlier, the only way the value of a function(and match is a function) can be fetched at the beginning of the query, before the first row is returned. While it would be possible to extend the columns with one for *k* in the same way as for distance, it would in no way be a property of the row and therefore a poor match for the SQL mindset.

3.1.3 Indices

Since this project is also an experiment to find good indices to use in SQLite it was necessary to support more than one index in SimiLite. Different indices will also fit better for different purposes; some might be better on kNN-queries with low k, while others might work better when many elements are returned. The cost of the metric will also be a factor; if it is relatively cheap minimizing the number of calls might be sacrificed in order for the index to use a more efficient architecture.

3.1.4 Creation, insertion, update and delete

For setting up a specific metric index, for instance LAESA, the following statement is executed:

```
CREATE VIRTUAL TABLE table USING SimiLite(
        id, object, distance, query, ./laesa);
```

The four first arguments are the user-given names of corresponding columns of the table. The fifth argument is the path to the index to use.

Insertion and delete works as expected, but only the id and object columns are accessible:

```
INSERT INTO table(object) VALUES (...);
DELETE FROM table WHERE id = 1;
```

Deletion must be supported in the index in use for SimiLite to support it. Any SimiLite functionality for the behaviour (for instance with a deleted flag) would fail for kNN-queries.

3.1.5 Extended syntax

Once a metric index can be constructed for one index it is easy to imagine the extension: Constructing an arbitrary table with a metric index on the given columns. This would make the table as close to an ordinary SQL-table as possible:

```
CREATE VIRTUAL TABLE image USING SimiLite(
        id INTEGER PRIMARY KEY,
        image_data BLOB,
        description VARCHAR(256),
        url VARCHAR(64),
        METRIC INDEX ON (description, ssstree),
        METRIC INDEX ON (image_data, laesa)
        );
```

This, however, would require the query-object and distance columns to have different names(possibly given by the user), and would make queries less intuitive:

```
SELECT id, image_data_distance, description
FROM image
WHERE image_data_query = _ AND image_data_distance < 1.0;
```

One would also have to create a query-plan for index queries on more than one index:

```
SELECT id, image_data_distance, description_distance
FROM image
WHERE image_data_query = _ AND image_data_distance < 1.0
AND description_query = _ AND description MATCH 'knn:10';
```

It is not straightforward to handle such queries, and an attempt would be outside the scope of this project. In addition it would not be any more efficient than splitting all the data into separate tables, as that is what would have to be done by SimiLite anyways.

For these reasons it was decided to keep the syntax as it is, rather than use poorly reasoned syntax and semantics. This should instead be the goal of future work.

3.2 LAESA

The design of LAESA at the beginning of the project was one table containing the pivots:

```
CREATE TABLE *_pivots (
        pivotId INTEGER PRIMARY KEY,
        pivotObject BLOB NOT NULL
        );
```

The object is stored in case the entry is deleted from the table. While the object cannot be a valid match then, the pivot will not become invalid. For each pivot one table containing the distance between the pivot and the objects were used:

```
CREATE TABLE *_pivot_p (
        id INTEGER PRIMARY KEY,
        distance FLOAT,
        UNIQUE (distance , id)
        );
```

the id is a reference to the id of the object, the distance is between the object and pivot_n ,and there is an index to quickly be able to find the relevant section. The query for doing pivot filtering in this instance is an n-way join for n pivots, which seemed like a lot of overhead:

```
SELECT *_pivot_1.id FROM *_pivot_1
JOIN *_pivot_2 USING(id)
JOIN ...
JOIN *_pivot_n USING(id)
WHERE *_pivot_1.distance >= _ AND *_pivot_1.distance <= _
AND *_pivot_2.distance >= _
AND ...
AND *_pivot_n.distance <= _;
```

The original motivation of the query was that each table was indexed and one could quickly locate the relevant entries and merge them, as shall be shown in the testing chapter however, this was not very efficient.

The design was then changed to contain all the distances in a single table:

```
CREATE TABLE *_pivot_distances (
        id INTEGER PRIMARY KEY,
        d1 FLOAT,
        d2 FLOAT,
        ...
        dp FLOAT,
        UNIQUE ( d1 , d2 , ... , dp )
        );
```

This allowed a much more efficient query to be used for the filtering:

```
SELECT id
FROM *_pivot_distances
WHERE d1 <= _ AND d1 >= _ AND  d2 <= _ AND ... AND dp >= _;
```

Instead of a join this will just be a scan through the table. Because of the index on the distance this will hopefully not be too many elements, but if d1 is not a good filter it will be equivalent to a linear scan.

3.2.1 Construction

Construction must take into account that not all elements are inserted at once and therefore when selecting pivots the choices will be limited. This requires a strategy for how many pivots should be used and when they are chosen. As mentioned in the background the choice of pivots is usually static. This, however, does not fit well within a dynamic table. Instead pivots are created dynamically once the table reaches certain sizes, but with decreasing frequency. A good growth to use for this is probably the logarithm ($p \in O(\lg n)$) and is what SimiLite uses. While it would be possible to require the user to decide the number of pivots to use beforehand, this is not very user-friendly.

As mentioned earlier pivots have good filtering power if they are far apart. One way to achieve this is to always add the object furthest away from the existing pivots as the new pivot, or equivalently; the object furthest away from its nearest pivot. This is the approach chosen in this project, and to aid this an extra table was added:

```
CREATE TABLE *_nearest_pivot (
        id INTEGER PRIMARY KEY,
        distance FLOAT,
        UNIQUE (distance, id)
        );
```

This table stores the distance from each object to the nearest pivot. The object of interest will simply be the entry with the maximal value, and this is an efficient query with the index. This strategy was adapted from [12]. To maintain the table its values will be updated whenever a new pivot is inserted.

To avoid costly ALTER TABLE statements whenever a new pivot is added the *_pivot_distances table is generated with a preset number of pivots. At time of writing this is set to 32 as few databases will be over 4 billion objects. If they are, a recompile and an ALTER TABLE on the existing index is everything required to migrate.

The cost of an insert will in the normal case, when no new pivot is added, be $O(p + \lg n) = O(\lg n)$. This is because the distance to each existing pivot must be calculated, and then an entry must be inserted into *_pivot_distances and *_nearest_pivot.

The cost will be a bit higher when a new pivot is added. In this case the distance to each existing object must be calculated, the corresponding entry of *_pivot_distances must be updated, and the *_nearest_pivots potentially be updated: $O(n \cdot \lg n)$. Amortized analysis shows it will be $lg(n)$ for each insertion. Between two pivot insertions, the first at k elements and the second at n, there are $n - k$ normal insertions, and one pivot insertion:

$$\begin{array}{crr} & (n-k) \cdot \lg n & \text{insertions} \\ + & n \cdot \lg n & \text{pivot-creation} \\ = & (2 \cdot n - k) \cdot \lg n & \text{total} \end{array}$$

Amortized this is:

$$\frac{((n-k)+n) \cdot \lg n}{n-k} = (\frac{n}{n-k} + 1) \cdot \lg n \tag{3.1}$$

By choosing logarithmic growth of pivots, as in this project, we have:

$$c \cdot \lg k + 1 = c \cdot n \equiv \lg k + \frac{1}{c} = \lg n \equiv k \cdot \sqrt[c]{2} = n \tag{3.2}$$

Combining (and taking into account that c will be constant) we have:

$$(\frac{n}{n - \frac{n}{\sqrt[c]{2}}} + 1) \cdot \lg n = \mathit{const} \cdot n \cdot \lg n \in O(n \lg n) \tag{3.3}$$

Despite low amortized time, the cost of insertions will be high for certain objects. One way of mitigating this problem is to split the cost over all insertions. Then the next pivot will be chosen at the same points as now, but its distances will be calculated during the insertions until the next pivot is chosen. By setting the number calculated to, for instance, two per insertion, the pivot will be ready when the next is chosen. It would also be possible to utilize the partial pivots when performing queries. The reason this approach was not chosen was that the cost of implementation was not considered worth it; most applications of metric indices seems to be setting up a large database, and thereafter only querying it.

Since there is little use for pivots for the first elements, and to have more choices later, the selection of the first pivot is postponed until a certain threshold, currently 32 objects. At this point the object furthest away from the first inserted object is chosen as the first pivot. The first object was used for simplicity.

3.2.2 Range query

The range query is as mentioned earlier the simple query:

```sql
SELECT id
FROM *_pivot_distances
WHERE d1 <= upper_d1 AND d1 >= lower_d1
AND d2 <= upper_d2 AND d2 >= lower_d2
AND ...
and dp <= upper_dp AND dp >= lower_dp;
```

The bounds for each pivot is calculated as $d(q, p) \pm range$ (look back to figure 2.3 on page 9 for a depiction of the situation). Any objects fulfilling this is turned over to SimiLite, which calculates and verifies the actual distance before it can be returned.

With m potential matches the cost would be a search for the first potential matching element, followed by checking the constraints for the next m elements; $O(\lg n + m \cdot p) = O(m \cdot \lg n)$. For the worst case queries it can be approximated that most elements are returned, in which case the runtime is $O(n \cdot \lg n)$. In this case a linear-scan would be better. This is currently not discovered, but could potentially use the statistics from SQLite to make the choice.

As can be seen this index will be very dependant on the filtering power of the first object, which might not be the best, therefore this choice was experimented a bit with as shall be explained in section 5.1.

3.2.3 kNN query

The kNN query for LAESA needs to iterate manually over every element and is therefore not ideal, but no other option was found because the query will change continuously. It starts as usual by calculating the distance to the pivots. Then, iterating over the table, elements are added to a max-heap until k elements have been found. The heap uses the distance as key, and the key of the top-element is an upper bound for the radius required for the equivalent range-query. For the remaining elements the same bounds check as for range-query is used to filter away objects, and the ones which pass are added to the heap if the actual distance is closer than the current max. The previous max will then be removed, and a new upper bound is chosen.

Since the checking is done in the index instead of in the SQLite core, control will cross the boundary more often, meaning that performance will not be as good.

3.2.4 Deletion

Since LAESA has a very simple structure it is only needed to remove an entry from *_main, *_pivot_distances and *_nearest_pivot to delete it. If the object should happen to be a pivot it will not affect anything; the pivot is still valid for distance comparisons, and it can never be returned.

3.3 SSSTree

The SSSTree paper specifies that when a cluster is split up non-centers are inserted into the bucket of the nearest center, but the centers themselves exist only in the internal node. For the implementation used in SimiLite this requirement was relaxed and the centers are inserted into their own bucket. This means that the number of leaves will be slightly larger and centers might be identical on consecutive levels. This might in turn yield worse performance, but it simplifies the initial implementation somewhat since only the buckets may contain returnable objects. These are queried anyway and the centers can therefore be ignored.

The table structure chosen for the SSSTree implementation was one table to keep track of the structure, and another to keep track of the objects. The structure is kept in the table *_clusters which is defined as follows:

```
CREATE TABLE *_clusters (
        id INTEGER PRIMARY KEY,
        parent_id INTEGER NOT NULL,
        center BLOB NOT NULL,
        covering_radius FLOAT NOT NULL,
        count INTEGER NOT NULL
        );
```

Here the id is the id of the cluster and it will be identical to the parent_id of the children. The blob of the center object is stored for quick access and deletion protection. The covering_radius is the distance from the center to the farthest leaf-node, and the count is the number of leaf nodes attached IF the object is a bucket. This is simply to avoid having to count the number of children whenever the database is opened, and it is used for knowing when a bucket must be split.

The table for the objects looks like:

```
CREATE TABLE *_main
        id INTEGER PRIMARY KEY,
        object BLOB NOT NULL,
        parent_id INTEGER,
        distance_from_parent FLOAT,
        UNIQUE(parent_id , id , distance_from_parent , object)
        );
```

The first two fields are the default for the main table, and therefore this structure can replace the table constructed by the main index module, and still be queried by it. This allows increased efficiency for the queries. The parent_id field is set to the cluster_id of the bucket this object is contained in, and distance_from_parent is naturally the distance between the center of the bucket and the object. Storing this distance turns the center into a limited pivot, and allows some rudimentary filtering which is described in detail later.

For the index the actual index is on parent_id since this is what is queried after and the id is there to make it actually unique. Distance and object is there for the clustering effect. As mentioned SQLite will use values if they are accessible in the index and this will be a bit more efficient. Clustering the object here means the blob will be stored twice, and this might be unacceptable if object is big or if it is a large database. This is simple to change however, and it was used like this in the smaller runs to be able to compete a bit more evenly.

3.3.1 Construction

The construction of an SSSTree is quite dynamic in nature, so there was little difficulty in adapting it to a SimiLite index. One freedom taken which differs from the original design is that a root object is allocated. This makes sure there are no special cases to handle in the code, and allows an estimate of the global covering radius.

The first object inserted becomes the center object of the root, and is inserted itself. Until the bucket is full new objects are simply inserted as leaves until the bucket reaches the threshold. At this point a procedure similar to the one described in SSSTree is used to locate new centers. The first object is the first center. The remaining elements is compared to all the current centers one by one. If one is further away than $2 \cdot \alpha \cdot covering\_radius$ it is added as a center. After this the objects are iterated over again and inserted into the closest bucket. This last part is slightly different from the one in the paper in that there the objects are inserted into the closest cluster if it is itself not a cluster while iterating. By waiting until all clusters have been chosen each object has more choices for the closest, and will therefore not be worse while it could yield a more efficient structure.

Once a tree has more than one cluster an insert will consist of comparing the new object to all cluster centers in a cluster, and recursively insert into the closest one. The distance from parent is of course updated if required.

The worst case construction for the SSSTree would be when every insert triggers a cluster split with only two new clusters, and one cluster only receives its own center. The structure will then be like a linked list and insertions are $O(n)$. This, however, is a very unlikely case, and under the assumption that the longest path from root to leaf is $O(\lg n)$ the result is better. Each insert will in this case need to locate its leaf in $O(\lg n)$, and then potentially split a bucket. If this is required the cost is $O(k^2)$ where k is the max number of elements in a bucket. The worst case occurs if every element forms new cluster. The total is $O(\lg n + k^2) = O(\lg n)$ since k is constant. As can be seen the runtime will be strongly tied to the distribution of the objects, and potentially the order in which they are inserted.

One advantage over LAESA is that the bound holds for any insert, and one will therefore not risk a suddenly expensive operation.

3.3.2 Range query

For a range query the query-object is compared to all top-level cluster centers. If there is any overlap between the covering radius of the query and the cluster (see figure 2.6 on page 11) it needs to be investigated further and is placed on a stack. Once all the clusters containing possible matches has been put on a stack an object is popped and the procedure is repeated recursively. Once a bucket is reached the following query is performed:

```sql
SELECT id, object FROM *_maij
WHERE parent_id = _ AND distance_from_parent >= _;
```

Any of these objects are possible matches and must be checked to see whether they are within the range or not. This ends up becoming a depth first search and ends when the stack is empty.

The distance_from_parent comparison needs an explanation. It is not really necessary as it will only filter away guaranteed misses, but since it avoids some metric calculations it was considered a simple way of increasing the performance of the index. The actual filtering can be considered as a variant of the LAESA filtering, but with only a single pivot.

The runtime is highly dependant on the number of clusters queried. Since the cost per cluster is constant when the maximum size is constant and $|clusters| \in O(n)$, this is the only operation considered. If every cluster is queried the cost is $O(n \cdot \lg n)$ since querying a cluster is a search followed by a constant number of steps. But for a relatively low radius most clusters can be discarded, and the runtime will be much lower.

3.3.3 kNN query

kNN-queries is not mentioned in the original paper, so a custom strategy had to be done. The kNN query for SSSTree makes good use of the tree-structure by using a search similar to the range query, but initially without a range constraint. It begins by calculating the distance to the cluster centers. Instead of discarding those outside the range, however, all the clusters, along with the distance, are stored in a min-heap with distance as the key. Then the top-element is popped from the heap, and the procedure is performed recursively on all the child-clusters. This is continued until a bucket is found, always picking the "closest" cluster. Once a bucket is found the elements contained are also put into a heap, but this is a max heap and is size-limited to k elements. This makes sure that it is easy to replace the furthest element with a closer one, and that it is efficient to switch it out.

This continues until there are k elements in the heap. At this point there is an upper bound for the distance to the k'th nearest element(the distance of which would be the radius required for the equivalent range query), and a filtering process can be started. The filtering process is pretty much identical to the above, but now that a max-range exists the same techniques used in the range query can be reused: The distance_from_parent constraint can be used as basic filtering and a cluster is only visited if there is overlap between the regions. The upper bound on the range is updated whenever an object is found that is closer than the current top of the heap.

When there are no more clusters on the cluster-heap the search is done, and the elements in the heap can be returned. The heap could potentially contain every cluster at some point (if there is only one level) so its size is $O(n)$.

The runtime will be the same as for the range query: Worst-case $O(n \cdot \lg n)$, but highly dependant on the k and the structure.

3.3.4 Deletion

The simplest way of implementing delete is to remove the object from *_main, which was chosen here ($O(\lg n)$). No errors come from this, but the tree might not be performing optimally. For instance, if the object deleted object is an outlier in the cluster, the new covering radius would be much smaller, and queries would visit the node even if unnecessary (another example is if all the leaves were deleted). This should, however, not be a big problem.

Since the cluster-centers themselves cannot be returned it is not necessary to remove them. If a strict SSSTree was needed it would be enough to mark a cluster-center as deleted to continue supporting correct deletion, and this should not impact performance in a very noticeable manner.

3.4 Test framework

The testing of the project in the previous iteration was incomplete and only performed one simple manual test of the performance, and a slow version for verification. It was expected that a good deal of testing was required for this project, therefore some effort was put into making a good basis for a testing framework. It the hope that this could be used as an aid in creating and optimizing indices and in comparing different types of indices easily against each other. Some initial high-level requirements were for it to get a graph of the results, easy scripting of the testing process and the possibility of comparing to a reference implementation.

Candidates for a reference was MSL and MoBIoS. MSL was chosen since there was already some experience with it, it had support for many more indices and datasets, and it was implemented in C instead of JAVA and should therefore be a bit faster.

Since it had to be easy to script the tests, python seemed like a good choice. It was something the author already was familiar with, is easy to program with and good graphs can be produced easily. One drawback, however, is that running the querying and index construction within python would be very disadvantageous for the project compared to MSL. Python is simply so slow that the times would be unfair.

MSL uses one program for constructing and one for querying the database and everything is in C. A similar approach was adopted for SimiLite; a wrapper in C was constructed which can either construct or query the database. The same input and output formats were chosen for easier implementation. A bonus is that it makes it easier to gather statistics about the run because it is entirely contained in the process.

3.4.1 Timing

An important part of the testing is locating where the time is spent, and it was wished that this could be displayed by the framework. For instance could a bad implementation spend more time calculating metrics than the total for similar indices, however that fact would be hidden. A possibility would be using a profiler, but that seemed like it would be unnecessarily complex to do automatically.

In the end it was decided to explicitly time the interesting sections of the code to give a good indication of time spent, and use a profiler to inspect more specific behaviour if needed. The interesting sections are: the distance computations, the layer above SimiLite (input/output manipulations, SQLite translation), SimiLite, the chosen index and finally the cost of using the shadow tables. One way of timing these sections is checking the time before and after each function call, and add it to a running total for the corresponding operation.

The end result looks something like in Figure 3.1. The smaller, red bar at the end signifies the difference between the sum of the various components and the actual CPU-time measured.

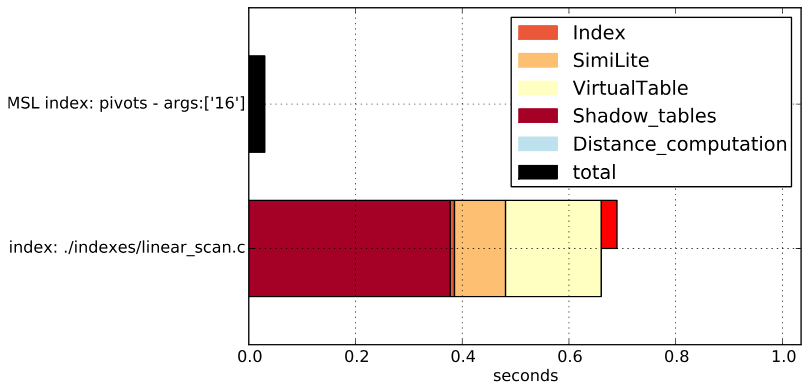


Figure 3.1: Example of comparing different indices with the test framework.

3.4.2 Syntax

The framework itself is based around instances of classes representing the different objects; the datasets (DataSet), the query-sets (QuerySet), the MSL indices (MSLIndex) and the SimiLite indices (SimiLiteIndex). To set up a test run a DataSet and a list of indices to test are needed:

```
nasa = DataSet(
    metric_space_name='vectors',           # Type of metric space
    instance_name='nasa',                  # Name of data set to use
    index_elements=40000,                  # Num objects to index
    path_to_metricspaces='./metricSpaces',# Path to directory
    radius=0.2,                            # Radius for the queries
    count=500                              # Num elements in query
)

laesa = SimiLiteIndex(
    lib_path='./laesa.so',                 # Path to binary file
    )

msl_pivots = MSLIndex(
    path_to_metricspaces='./metricSpaces',# Path to directory
    index_name='pivots',                   # Which index to use
    index_args=['17']                      # Num pivots to use
    )
```

When this is set up the test can be run simply as:

```
run ( indices=[laesa, msl_pivots], data_sets=nasa )
```

Only the binary file for the index is needed when setting up the SimiLite index, but it is possible to give the source file and let the framework handle compilation. At this point it is possible to give additional compilation options, which makes it easy to compare different versions of the same index by enabling and disabling features with the use of macros. A motivating example can be finding the optimal value for α in SSSTree:

```
ssstree_list = [SimiLiteIndex(
    source_path='./indices/ssstree.c',
    compile_args=['-DALPHA=' + str(x)],
    ) for x in [0.3, 0.35, 0.4, 0.45]]
```

Chapter 4

Implementation

SimiLite is written in C, partly because SQLite is written in C, but also to attempt to get as much speed as possible from the execution external to SQLite. It is a researchers implementation rather than a full product in that it handles errors badly and has not been extensively tested for edge-cases. This is not because it would be particularly hard to do those things, but because the time was better spent elsewhere. If this is a viable concept, these things can easily be dealt with later. The internal SQLite memory allocation subsystem (*sqlite3_malloc*, *sqlite3_realloc* and *sqlite3_free*) was used to be as compatible with SQLite operation as possible.

This chapter in turn discusses the more interesting implementation details of SimiLite, LAESA, SSSTree and the testing framework, and will be of special interest to anyone who plans to work with the source code.

4.1 SimiLite

This source code described in this section was mostly written in the previous project.

The struct representing a SimiLite instance looks like:

```
typedef struct DiTable
{
    sqlite3_vtab base;          /* Base class, used by SQLite */
    sqlite3 *db;                /* The database connection,
                                   needed for shadow tables */
    char *zDb;                  /* Logical database name, needed
                                   for resolving potential table
                                   name ambiguity */
    char *zName;                /* Virtual table name, needed for
                                   querying the correct tables */
    IndexState *pState;         /* The index used for this table */
    metric_t metric;            /* Function pointer to the metric
                                   for this index */
    const IndexModule *pModule; /* Consists of functions and
                                   properties defining the behavior
                                   of the index */
    sqlite3_stmt *select_by_id; /* Used for faster queries */
} DiTable;
```

The *db* object is used in running the queries on the database, and *zDb* and *zName* are the names used to identify the correct shadow tables. The first describes the context of the database and will usually be "main", but it can be "temp" for temporary databases, or a given name if the database has been attached to current main. The *IndexState* and *IndexModule* will be described later. The distance function is named metric and will also be used by the index module. Unfortunately the metrics used by MSL used the internal id of the objects as arguments, and is therefore could not be linked to directly.

xCreate in the index sets up the table *_main which stores the actual blob:

```
CREATE TABLE *_main (
        id INTEGER PRIMARY KEY,
        object BLOB NOT NULL
        );
```

To allow higher efficiency it is only added if an index has not already done so. It will, however, always assume that the id and object fields exist. If only the linear scan was required this would be enough, but more efficient indices will need additional data-structures.

xCreate or *xConnect*, whichever used, will use the *dl* library to dynamically load the indices. The index must have implemented a function named *similiteIndexInit* which returns a module representing the index (to be described later). This has been made dynamic to make it easier to test SimiLite with different compilations of the same index. From the user perspective it would be better to compile all the different indices into the SimiLite library. Since SimiLite can itself be recompiled and reloaded if necessary, the ability to dynamically insert a new index is not useful.

The xBestIndex implementation of SimiLite divides queries into 4 cases: no constraints (1), equality constraint on id (2), $</\leq$ on distance (3), or only a constraint on the query-object(4):

```
1: SELECT * FROM table;
2: SELECT * FROM table WHERE id = _;
3: SELECT * FROM table WHERE distance <= _ AND query = _;
4: SELECT * FROM table WHERE query = _ AND object MATCH 'kNN:10';
```

If there are no constraints a simple linear scan is used, and for equality constraint on id the constraint is just transferred to a similar query on *_main. Because of the index on id this will be much more efficient, and can be used for instance to find the distance between a given object and a query. Constraints on distance is converted to a range query and left to the index. A query-object is required in this instance, otherwise there is nothing to compare with. If there is a constraint on the query-object, the query can either be a kNN query, which is left to the index, or simply a listing of all the objects along with distance from the query, which SimiLite performs with a linear scan.

Any other situation is handled by performing a linear scan and letting SQLite check the conditions. In this case query and distance are illegal columns to select from.

The function xFilter positions the pointer according to the strategy chosen by setting up the query and calling xNext.

xRename was not implemented to allow use of prepared queries without needing facilities for recompiling. This should not be a great loss.

The other functions follow closely to the design, and a description should not be needed. Much of the structure and set up here was inspired by R*-tree and Fts3.

4.1.1 Index interface

The IndexModule contains the methods defining the current index in the same way as for the Virtual Table module. It is also loaded in a manner inspired by the virtual table module: the index contains a method which when called returns an IndexModule instance with the correct methods. SimiLite stores this and calls the methods at the appropriate time. The methods contained in IndexModule are:

setUp Sets up the state required to interact with the table and sets up the backing structures if the argument isCreate is set. This returns an instance of a subtype of IndexState which is used as an argument in the other method calls.

cleanUp Frees the state and destroys the backing structures if the argument isDestroy is set.

setUpRangeQuery Sets up the state required to perform a ranged query on the table. This must initialize a subtype of the struct QueryPointer which is used as an argument to queryStep. The range is an argument.

setUpkNNQuery Same as above, only for a kNN query. The *k* is an argument.

freeQuery Frees up the QueryPointer and related structures.

queryStep Moves the pointer to the next possibly matching row. The return value signifies whether it was a valid row or if the previous row was the last one. queryStep must return the id of the possible match, and may for efficiency reasons return blob and distance information if available, otherwise these will be fetched by SimiLite when required.

insert Informs the index of a new row that has been inserted into *_main. The id and blob of the new insert are given as argument. .

delete Informs the index of a pending deletion. The function can decide to make SimiLite also remove the entry from *_main. This function pointer can be set to NULL if deletions are not supported by this index.

It is the purpose that the index extends IndexState and QueryPointer with index specific data. At the moment they simply contain pointer to the connected SimiLite instance and IndexState respectively, but can easily be extended with other generic data. For instance statistics and properties which can be queried/modified with SQL. It is wise to keep this away from the SimiLite state in case it at some point becomes possible to use more than one index on a table.

As mentioned queryStep needs only move the query to a possibly matching row, this is done to leave the index with as low a burden as possible, since SQLite will check the distance constraints by itself. As a consequence the range query needs only handle the case $\leq$, and SimiLite will filter away non-matches if the actual constraint is $<$. This also means that a linear scan can be, and was, implemented by simply returning all rows in the table one by one.

4.1.2 Errors

During implementation most bugs has been in a few general categories: writing SQL-statements correctly, using the SQLite interface correctly and mistakes with the allocation and freeing of memory. The first is mostly a problem because, while errors from SQLite sometimes can give a descriptive error message, this has not been well propagated to output from SimiLite. The solution is to debug where the error happens and fetching the message manually. Since there are few operations done on SimiLite this is usually not hard. An example on using the interface correctly bug is using the function *sqlite3_prepare* instead of *sqlite3_prepare_v2* in a single instance. This was a somewhat hard to find bug.

If the mistakes performed until now are representative it seems like most bugs will be easily visible, but not necessarily easy to find.

4.2 LAESA

Below are the structure of the state for the LAESA implementation. Only a little state is added: the number of elements, for calculation of when to add a new pivot; the cached pivots, to avoid fetching them again for every query; and various prepared statements, to avoid creating for every usage. By including base first as in *LaesaState* and *LaesaPointer*, the structs will behave as the base class when used through a base class pointer.

```
typedef struct LaesaState
{
    IndexState base;            /* Base class, used by SimiLite */
    sqlite3_int64 numElements;
    sqlite3_int64 numPivots;
    sqlite3_stmt *insertInPivot;/* This and the next are prepared
                                   statements used for efficiency*/
    sqlite3_stmt *insertNearest;
    PivotObject **ppPivots;     /* The cached pivot objects */
} LaesaState;

typedef struct LaesaPointer
{
    QueryPointer base;          /* Base class, used by SimiLite */
    sqlite3_stmt *statement;    /* select statement for range query
                                  NULL if kNN query */
    MaxPriorityQueue knn_heap;  /* Contains k Nearest Neighbours if
                                   a kNN-query, NULL otherwise */
} LaesaPointer;
```

sInsert calculates the distance to all existing pivots and adds the entry to *_pivot_distances. A new pivot is chosen if the next step has been reached, in which case distances to all previous objects are calculated. For locating the first pivot a control variable called *FIRST_PIVOT_THRESHOLD* is used to make sure there is a better basis for making the choice. *PIVOT_LIMIT* can similarly be used to set the maximum number of pivots which will be used.

As shall be mentioned in the chapter about testing; a lot of different options was attempted in improving LAESA. To be able to see the effect of the modification it was made possible to turn features on and off with C preprocessing macros. They are of the form:

```
#ifdef USE_INDEX
# define CREATE_STATEMENT_PIVOT "CREATE TABLE %Q.'%q_pivot_%d'(id ↩
    INTEGER PRIMARY KEY, distance float, UNIQUE (distance, id));"
#else
# define CREATE_STATEMENT_PIVOT "CREATE TABLE %Q.'%q_pivot_%d'(id ↩
    INTEGER PRIMARY KEY, distance float);"
#endif
```

This make it easy to have features that can be turned on or off, but requires a certain care when implementing. Some of the macros will interact and several cases may have to be evaluated. Since the code is riddled with such macros maintenance can be difficult, but it is only necessary to support for the duration of this project. The inferior features can easily be removed afterwards.

One especially clever, and perhaps therefore unintuitive, use of these macros came about in implementing the sorting of the pivots according to distance from the query. The existing version iterated through the pivots, calculated the distance to the query and inserted the constraint into the query:

```
// The following code is greatly simplified
for ( int i = _; i < _; i++ ) {
    dist = metric ( ... );
    add_to_query ( i, dist, ... );
}
```

The sorting was implemented by splitting the for-loop and sorting in-between:

```
// The following code is greatly simplified
for ( int i = _; i < _; i++ ) {
    dist = metric ( ... );
#ifdef SORT
    array[i].dist = dist;
    array[i].id = i;
}
qsort ( array, ... );
for ( int j = 0; j < _; j++ ) {
    int i = array[j].id;
    dist = array[j].dist;
#endif
    add_to_query ( i, dist, ... );
}
```

This has the desired effect, but can be very confusing at first.

For the heap used in kNN code from [28] was adapted (this was also used for the heap in SSSTree kNN).

4.3 SSSTree

The standard structs were implemented as follows:

```
typedef struct SSSTreeState
{
    IndexState base;
    cluster_t *root;                          /* Top of the tree... */
    sqlite3_stmt *insert_cluster_statement;/* This and the
    sqlite3_stmt *radius_update_statement;     following are used
    sqlite3_stmt *insert_object_statement;     for efficiency. */
    sqlite3_stmt *select_objects_statement;
} SSSTreeState;

typedef struct SSSTreePointer
{
    QueryPointer base;
    cluster_stack_t *stack;             /* The stack of clusters with
                                           potential matches */
    sqlite3_stmt *running_statement; /* Used in querying buckets */
    object_t query_object;              /* Cached query object */
    double range;                       /* Cached range of query */
    int started;                        /* Signifies whether the query
                                           has started, set to true in
                                           the first sQueryStep */
    MaxPriorityQueue knn_object_heap;/* Contains k Nearest
                                           Neighbours if a kNN-query,
                                          NULL otherwise */
} SSSTreePointer;
```

The *cluster_t* struct is a cached version of the rows in the shadow tables. Similar to LAESA this allows more efficient queries, perhaps even more in this case because there are many more clusters than pivots in LAESA. The difference was never tested as SSSTree was designed to be cached from the beginning.

```
typedef struct cluster_type {
    sqlite3_int64 id;
    float covering_radius;
    object_t center_object;
    int count_children;             /* A cluster with 0 children
                                       is a bucket */
    /* Double pointer to avoid having to copy
       the elements when resizing: */
    struct cluster_type **children;
    int allocated_children_size; /* Used to know when "children"
                                       must be resized. */
    int bucket_count;               /* If the object is a bucket
                                       it represents the number of
                                       elements in the bucket,
                                       otherwise it is undefined */
} cluster_t;
```

The size of children (and every other expanding list in SimiLite) is doubled at each step for amortized constant runtime $(1+2+4+...+n =$

$2 \cdot n - 1 \in O(n)$).

The tree of clusters is created in *setUp* by iterating through the list of clusters in order of id. Each entry is inserted into an array based on id, and into the child list of its parent cluster. The parent is fetched by id from the same array. Since a parent will always be added before its children, the id will be lower, and therefore it will already have its entry inserted into the array.

Operations which were similar was extracted into methods:

```
cluster_t *new_cluster ( SSSTreeState *pSState, cluster_t *parent, ↩
    object_t object, sqlite3_int64 id );
```

The method *new_cluster* was used during loading of the tree graph to set up and correctly link the *cluster_t* objects and when adding a new cluster. If creating, the id will be set to a special value which indicates that the new cluster should be inserted into the shadow tables. The id of the new cluster will be set in the returned object.

```
void compare_to_cluster_children (
        SSSTreeState *pSState,
        cluster_t *cluster,
        object_t object,
        int break_on_noncluster,
        int *possible_cluster,
        sqlite3_int64 *nearest_cluster_child_id,
        double *nearest_cluster_dist)
```

The method *compare_to_cluster_children* is used when the correct bucket to insert a new object in is located, or when a bucket is split to check if the object should be a new cluster.

sInsert calls the method *insert_into_cluster*. This method locates the closest center and recursively inserts the object. If the object is inserted into a full bucket it is split, and insert_into_cluster is again used to insert the objects into the new clusters.

To control the cluster size there are the two usual control variables as macros; ALPHA and CLUSTER_SIZE_THRESHOLD. Because there are at most a constant number of objects in a bucket, there will be at least $\frac{n}{CLUSTER\_SIZE\_THRESHOLD}$ or $O(n)$ clusters. This means that a constant fraction of the database must be in RAM, a solution which will not scale well. It would be possible to improve on this, for instance caching often used elements in a LRU style buffer with constant size, however, the need never arose during development.

With CLUSTER_SIZE_THRESHOLD set to 50, 100 MB of RAM will be enough for a database of up to 10 GB.

An interesting edge-case is when the same object is inserted so many times that a bucket will be split when only containing that object. The covering_radius will then be zero and every object will become a new cluster_center, and all the objects will be inserted into the first center. Since the objects were enough to split the bucket, it will need to be split again, and so on. One way to fix this would be to implement the SSSTree strictly as in the paper, where the cluster-centers are not inserted deeper into the tree. This problem was never seen in practice, however, and therefore not fixed.

4.4 Test framework

The functions of the framework is constrained the most by the MSL formats so first a discussion of these are in order.

4.4.1 MSL formats

There are three programs which must be used in order to utilize the MSL indices as intended; one for construction, one for generation of queries and one for execution of the generated queries.

For construction the following command is used:

```
build-"index_type"-"metric_space" dataset num_elements ↩
    index_filename (index_arg0 , ...)
```

The library generates one program for each combination of metric space and index.

The format of the dataset varies according to the space chosen. For vectors it is simply the vectors lined up one after the other with a little header (number of dimensions, which Minkowski distance is used) data at the beginning. For documents it is the path to a folder containing documents named with its own id. The *index_filename* is simply where to store the index once created, and the *index_args*, if any, are passed directly to the index itself and are used in the construction. For pivots this is the number of pivots to use. One point worthy of note is that the indices only store the index specific data. For accessing the objects themselves it stores a reference to the original dataset as a relative path. This means that the index cannot be queried from a different folder than where it was created, and that it can use the data directly, thereby avoiding some of the work.

Once the index has been constructed it must be queried. There are two programs used for this, and they communicate with a string describing the queries. The format is first a variable describing what kind of query it is, followed by a space-specific description of the query-object. For vectors it is the vector and for documents the id of the document. The query can either be the radius of a range-query, or the k of a kNN-query given as the negative k. A "-0" on a line by itself ends the querying. An examples is:

```
0.1,0.43,0.13,...
-4,0.29,0.88,...
-0
```

The command to generate the queries is:

```
genqueries dataset start num query_info
```

This will generate *num* queries between *start* and the end of the dataset. This can ensure that elements in the index are not also used as queries. The *query_info* parameter is either positive for radius or negative for kNN as explained earlier. The *genqueries* command is specific for each space, and they may therefore be slightly different, for instance in vectors there is an additional parameter, perturb, which will be added or subtracted to a random dimension of the object before the query is performed. This functionality has not been needed in this project.

The queries are fed into the last MSL command; query:

```
query-"index_type"-"metric_space" index_filename
```

As for construction, there exists one executable for each combination of space and index. The program executes the queries it receives from *stdin* on the index. The valid outputs are written to *stdout*, while the number of hits per query and some other statistics are written to *stderr*.

During attempts to compare MSL and SimiLite in low RAM conditions it was uncovered that MSL requires a certain amount of RAM (it keeps the entire index there), and will fail if it does not have it.

4.4.2 Timing macros

The layers chosen to track were: everything above SimiLite (the cost of using a virtual table), SimiLite itself, the index, the shadow tables, and the distance computations. To measure the time spent in the different layers, all activity crossing in or out of SimiLite is tracked. Timers are started by subtracting the current time and started by adding. The elapsed time will then remain $(-start + (start + elapsed) = elapsed)$. To avoid having to locate and add timing code to every return statement, *sqlite3_module*, the struct with the function pointers sent to SQLite, is filled with timing code wrapped replacements. For the functions below SimiLite a timer is simply started before and stopped after the function call. To make sure that only one timer is running the currently running timer will correspondingly be stopped and started. This was done by wrapping with a macro. Then functions could be timed with for instance:

```
TIME_INDEX (
        return_value = index_call ( ... );
        )
```

This would be turned into the correct sequence.

```
index_call_start ();
return_value = index_call ( ... );
index_call_stop ();
```

Here *index_call_start* would stop the timer for SimiLite and start the timer for the index and vice versa for *index_call_stop*. Doing it like this also makes it easy to remove all traces of the timing code from SimiLite in case it is no longer needed. The results from the timings is printed to *stderr* upon the closing of the database.

It was deemed inconvenient to require the index to use timings, but it can be a help to know exactly how much of the time spent in the index which is actually in the shadow tables. Therefore the index can also track these calls.

Since the index can call the metric function directly, but is not required to time it, this case must be handled differently. It is done by replacing the distance function with a wrapper-function which performs the timing.

To make it easy to see how much time is spent in a given code snippet a special timer is also available. This only tracks the time spent in the relevant section, and does not stop the running timer. This should be an aid in quickly deciding whether some code is worth optimizing.

4.4.3 SimiLite runner

As mentioned in the design section, the insertion and querying code was extracted into a separate program in order to avoid having the logic in the testing framework itself. This runner is greatly inspired by the equivalent MSL programs, but merged into one. There is one mode for insertion and one for querying, and the first parameter decides which:

```
similite_runner create dataset_format dataset index_filename ↩
    num_elements index_to_use inserts_in_memory(true|false)
similite_runner run_queries dataset_format index_filename verify(↩
    true|false) query_in_memory(true|false)}
```

The parameters are the same as for MSL where the names match. The parameter *dataset_format* is set to the metric space of the dataset (vector, documents...), *inserts_in_memory* (if true) creates the database in memory before writing to disk, but this seemed to have little effect. For querying *query_in_memory* is equivalent to *inserts_in_memory*, while verify sorts the output by id. This is used in the simple verification scheme used as shall be explained.

MSL does not provide a get-method or any other way of parsing the dataset for a given space, so functions had to be made for this manually. To enable support for more than one space this function is chosen dynamically based on the dataset parameter. The current implementation only has support for the vectors space, but since it should be a simple matter to add support for more.

The querying is done as in MSL with parsing the input from *stdin* to get the queries, and printing the matches to *stdout*. For this functions for parsing the query-object string into the binary object, and for printing a binary object to string, is needed. These are handled dynamically as for the above.

4.4.4 The framework

Tying all this together is the framework itself, which is, as mentioned, written in python. The job is simple: compile the indices if necessary, build one version of the database for each index, generate a query-set and finally query the indices. To run the external programs the python module subprocess is used. To time the MSL indices the program is run within the *time* program, which simply outputs the time spent to *stderr*. After the process is finished the output is parsed to retrieve the timing values from *time* and from the wrapper.

When preparing to query the indices *genqueries* is used to store the queries to file. Due to the difficulty of certain queries it is important to use the same queries for all the indices. For instance is it usually more time consuming if there are many objects within range than if there are none.

The graphs are created with the *matplotlib* function *broken_barh* as this was the only function found capable of displaying the timings as desired. One for each construction and querying time is made for each run.

The previous verification scheme was way too slow because the metric was computed in python (since python objects were the blobs), and this caused much boxing and unboxing. The new verification is integrated into the testing framework, and is simply done by comparing the output of a linear scan with the output from the index in question. The runner makes sure the output is ordered by id in this case. This linear scan has in turn has been verified with the MSL solutions, as these are presumed to be well tested.

The framework also support graphing the number of bytes read from or written to disk, but the results seemed somewhat unstable due to buffered files and such. Results from such tests were therefore not included.

Chapter 5

Optimizations

This chapter takes a look at attempts to make the indices more efficient. The tests have been run a few times to verify consistency and, if not stated otherwise, 500 small queries from the NASA dataset was used. All the indices have been verified to work correctly.

5.1 LAESA

The initial LAESA did not show satisfactory performance, therefore various solutions was attempted to improve on it. Some of these are detailed here, not all of them successful, but something can be learned even from failed attempts.

Cost of indexed values

It was observed that the index had a high construction cost, while not seeming to increase performance. On the contrary it deteriorated as can be seen in figure 5.1. It should also be mentioned that the construction time was significantly reduced (more than 50%) by dropping the index. This fact varied according to other parameters though, as there was significant improvement in using the index for the unified table approach (see figure 5.2) for the querying.

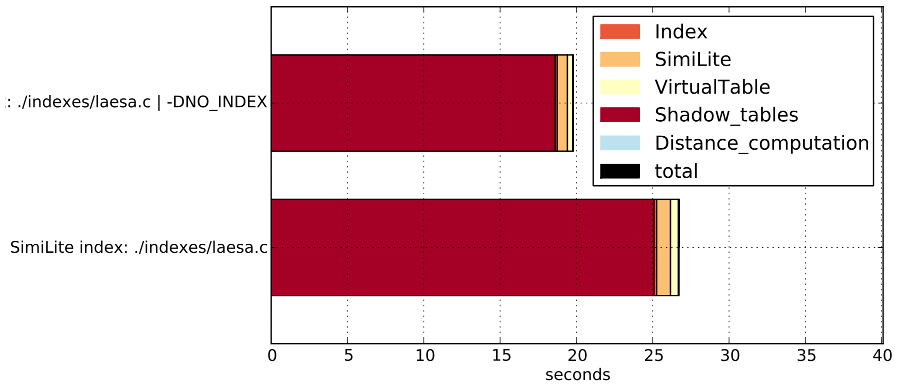


Figure 5.1: Comparison of using index or not for LAESA with one table per pivot.

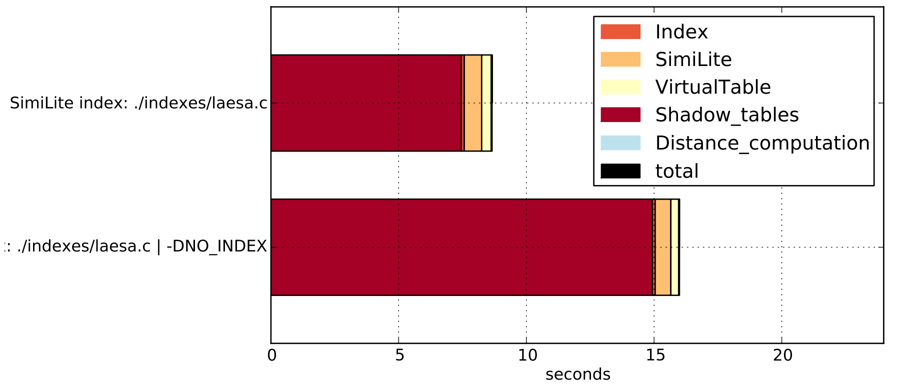


Figure 5.2: Comparison of using index or not for LAESA with one unified table.

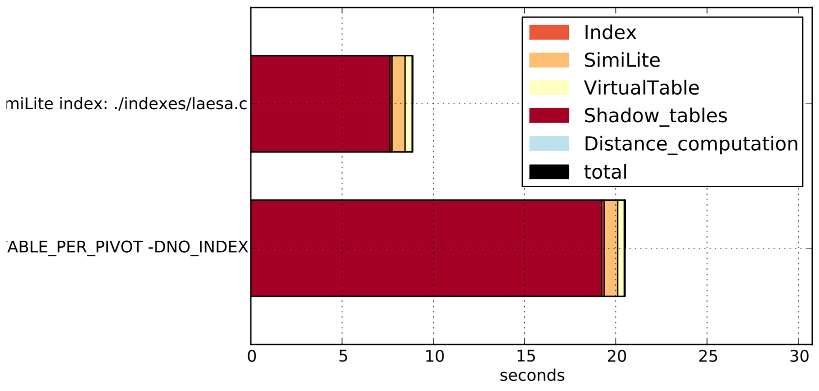


Figure 5.3: Comparison of using one or many tables for LAESA.

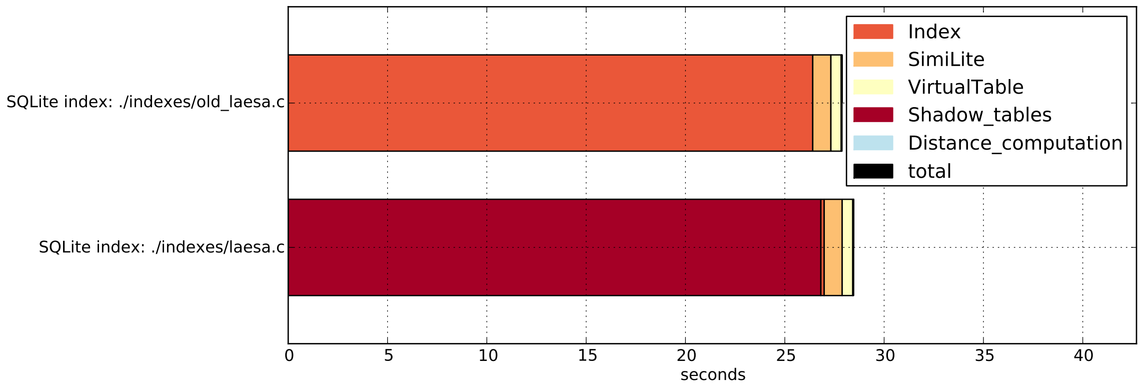


Figure 5.4: Comparison of caching pivots in LAESA with 500 small queries. Since the timings was never back-ported into the old LAESA, time that is actually spent in Shadow_tables and Index is accounted for under SimiLite.

One vs. many pivot tables

As mentioned the design was changed to utilize one shared table instead of one for each pivot. The results for the querying can be seen in figure 5.3, and, while not shown, the improvements to construction time was also significant (close to 50%). It is assumed that this is caused by the high cost of the joining, which did not confer any benefits.

Metric type

It was hypothesised that the choice of data type for the index could be a performance factor. Double was used initially as it seemed more accuracy would be advantageous. A side effect of this will be a larger table (twice the size when compared to float), which should take longer to work with. A switch to float was attempted since the numbers are approximate bounds they need not be accurate, but the modification had no effect. Even the database size was identical, and this proved to be because SQLite only has one internal representation for floating point; 64 bit[29].

Pivot caching

LAESA originally fetched the pivots each time a query was performed. While this should be a small part of the query, it is unnecessary and

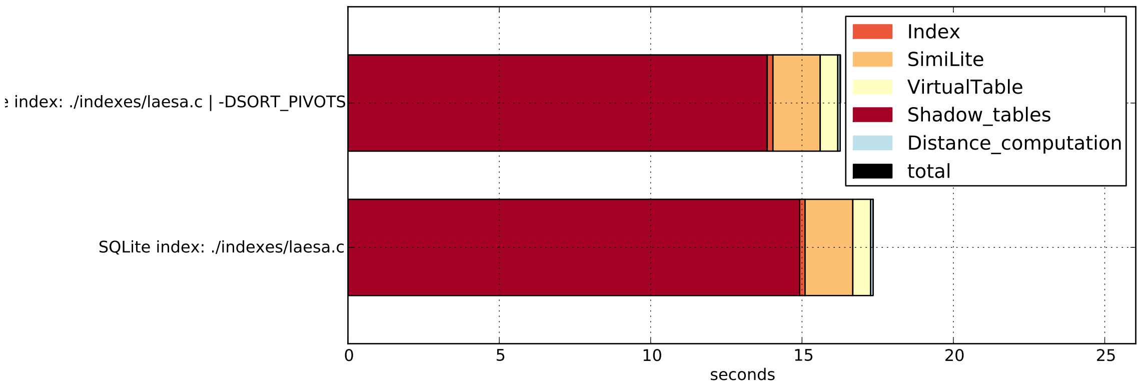


Figure 5.5: Comparison of ordering the pivot constraints by least distance from query first. Tested with 500 small queries.

they are therefore cached whenever the table is opened. Since this feature was not made switchable with a macro, it is compared with the old LAESA while having all the other improvements turned off. As can be seen in figure 5.4 this unfortunately had little to no effect.

Sorting of pivots

Another unfruitful idea was reordering the order of the constraints to apply the most constraining first. The motivation was that these would filter away the most objects, and therefore avoid checking many of the constraints. This observation, perhaps naively, assumes that SQLite has short-circuit logic, and that the constraints are checked in the same order as they are received in. The most constraining pivots are the ones with the least area of the "ring" (see figure 2.4a on page 2.4a). Since the width is the same for all of them, only the radius, or distance from the query, matters. By sorting the pivots according to distance and applying them to the query in this order the goal is achieved, but as can be seen in figure 5.5 it had only a slight impact. The effect disappears for larger queries, which is natural as most constraints will fail anyway.

Finishing the query

One possibility which is much friendlier with the memory hierarchy is finishing the query when starting, and caching all the results in a table. For later attempts the entries from this table can be returned. As can be seen in figure 5.6 there is an effect for large queries. For small

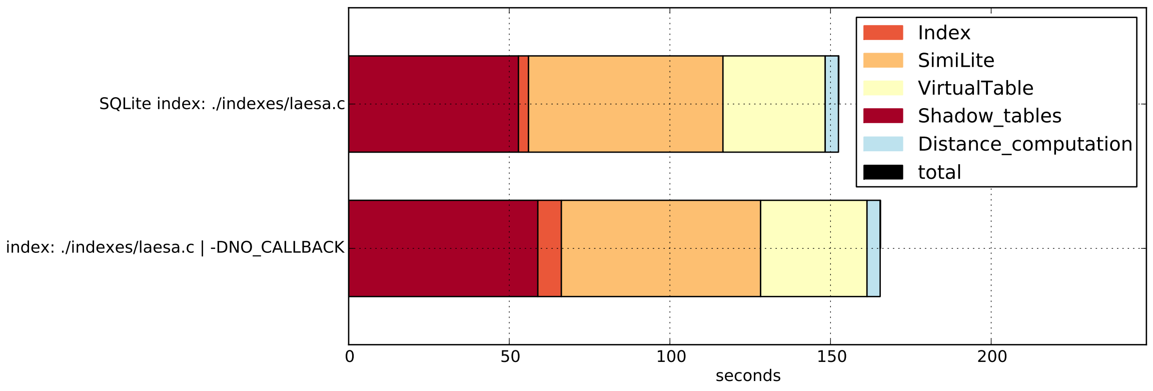


Figure 5.6: Comparison of fetching the results at the beginning instead of fetching the next each time. Tested with 500 large queries.

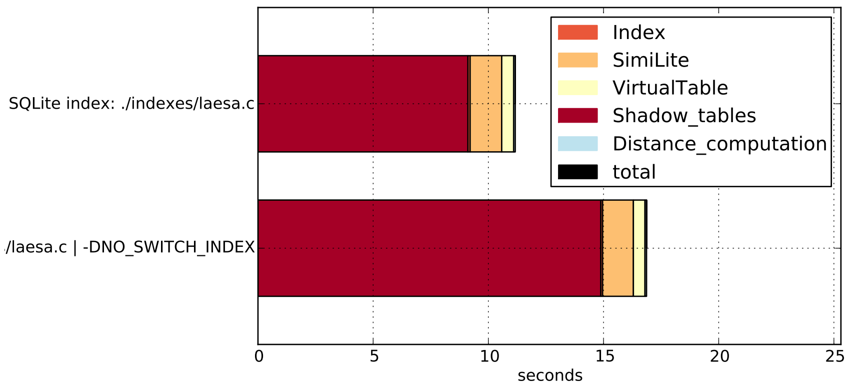


Figure 5.7: Comparison of querying time where the index on pivot_distances was reversed. Tested with 500 small queries.

queries, however, the improvement disappears. This can probably be combined with the sorting above to achieve some effect in both cases.

Reversing the pivots

As mentioned in section 3.2.2 the order in which the pivots were indexed was not optimal because the first pivots are chosen with fewer candidates. One way to improve on this should be to simply reverse the order in which they are indexed. Since the index is used before it is filled, it must be usable at all times. This is done by inserting NULL for the distance in a pivot if it has not been chosen yet. By setting these constraints on the query as well, the index will be usable:

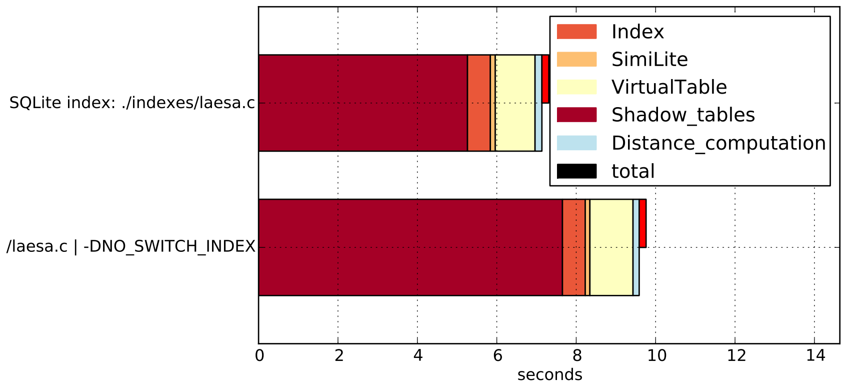


Figure 5.8: Comparison of construction time where the index on pivot_distances was reversed. Tested with 500 small queries.

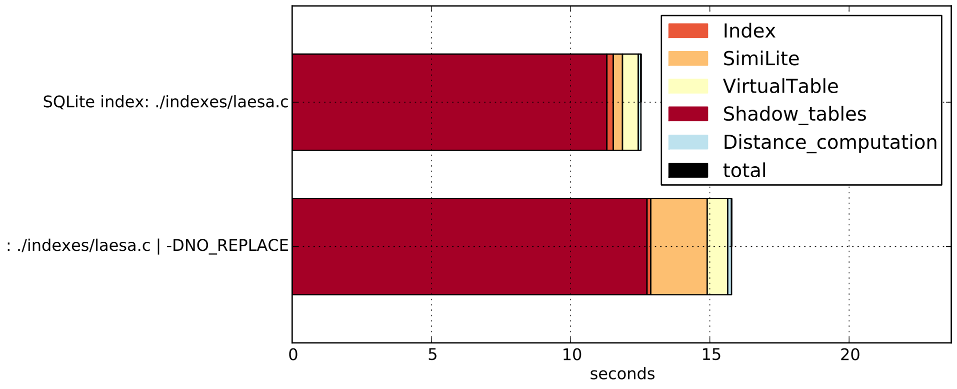


Figure 5.9: Comparison of query time where the *_pivot_distances table was merged with main. Tested with 500 small queries.

```
SELECT id
FROM *_pivot_distances
WHERE d1 <= upper_d1 AND d1 >= lower_d1 AND  d2 <= upper_d2 AND ...↩
     AND dp >= lower_dp AND dp+1 = 0.0 AND ...;
```

The results are displayed in figure 5.7. This is a significant speedup and backs the hypothesis. Interestingly the construction time is also affected by the change (figure 5.8). This is even though the entire index basically has to be rebuilt (sorted) for every pivot insertion.

Replacing *_main

During profiling it was observed that using the SimiLite facility for fetching the object by itself was costly. This is because the *_pivot_distances table does not contain object, and is therefore it has to be fetched from

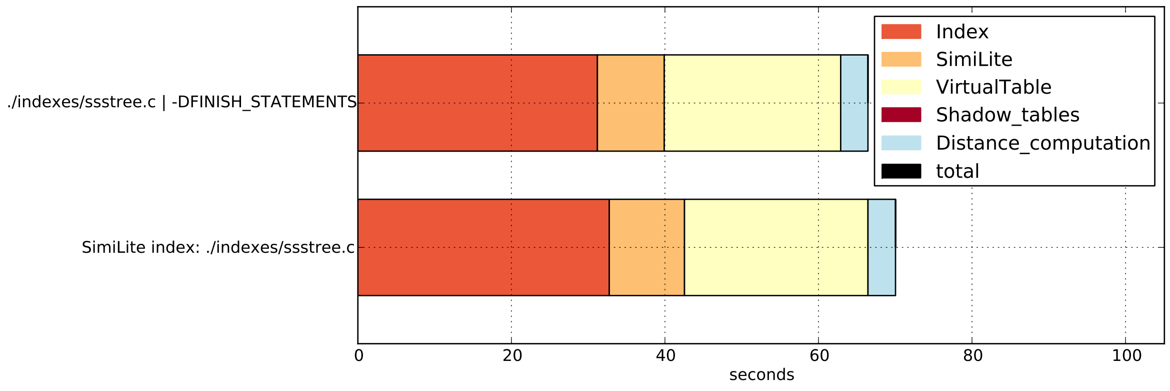


Figure 5.10: Comparison of fetching all matches each time a bucket is queried. Tested with 500 large queries.

*_main. The reason for the original separation of the data into two tables was to keep things as simple as possible until it was certain that everything worked correctly. After having implemented SSSTree, where the *_main table is replaced, it was decided to also attempt merging the two for LAESA. The results are in figure 5.9, and there is a clear advantage. One point of interest is that the SimiLite share has almost completely disappeared because it does not have to do the separate fetch now, additionally the time spent in the shadow tables was also decreased.

5.2 SSSTree

Not too many different versions of SSSTree was attempted, as it performed pretty satisfactory from the beginning, and many of the attempted improvements to LAESA was part of the initial implementation (unified table, caching).

Finish statements

Like for LAESA, finishing the query whenever started was attempted as this can give better memory. SSSTree should also have a slight benefit of "amortizing" the cost across the steps, since a bucket has maximum CLUSTER_SIZE_THRESHOLD results. The result can be seen in figure 5.10, and there was not really a significant difference. This is to expected, as the difference for LAESA was small.

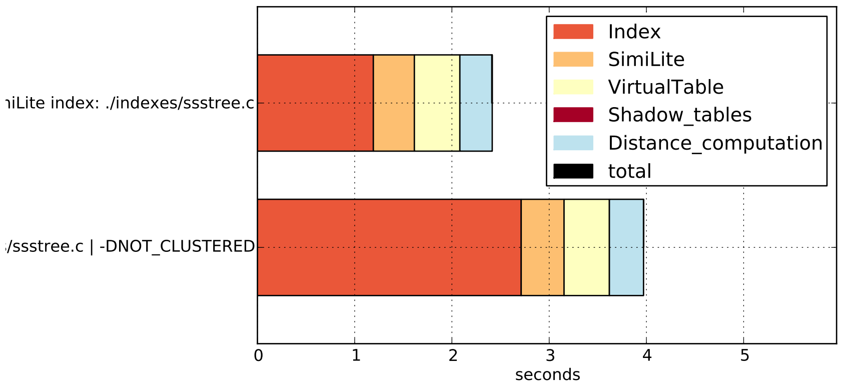


Figure 5.11: Comparison of fetching all matches each time a bucket is queried. Tested with 500 large queries.

Clustering objects

The *_main table in SSSTree used an index on (*parent_id*, *id*) for quick lookup of the children for a bucket. When iterating the objects , however, the object and distance is also needed, and this causes SQLite to make another lookup in the main B-tree of the table. This lookup can be avoided by clustering the fields into the index: (*parent_id*, *id*, *distance_from_parent*, *object*). The *id* is placed before the two other fields to avoid having to compare them, as it is unknown how efficient this is for SQLite, and it is still needed anyway.

Figure 5.11 demonstrates the results, and they are very significant (about 1/3 faster). It should be mentioned that the construction time is also increased somewhat by this change.

Pivot filtering

The SSSTree implementation in this project chose to include distance from the objects to its lowest parent, and used this as a primitive pivot filtering. The result is displayed in figure 5.12. The improvement is quite significant, nearly halving the time spent. Additionally the time spent on distance computations went down by about 1/3, meaning that this boost will stay, even if the cost of the metric increases greatly.

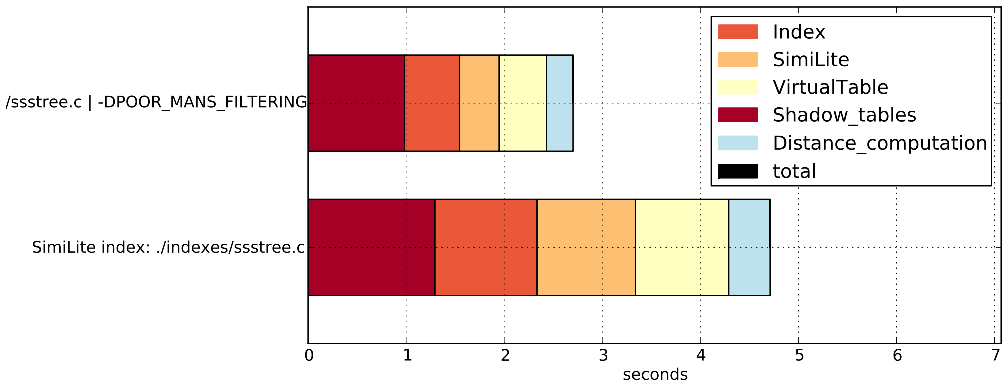


Figure 5.12: Comparison of using poor man's filtering or not in SSSTree.

Chapter 6

Analysis

An important question when considering this plug-in is the impact on performance of the SQLite wrapping, and that of the shadow tables. For queries where the cost of calculating metrics dominates, everything else will be insignificant and performance will be approximately equal with other semantically identical index implementations. The remaining cases then are cases where the cost of the metric does not dominate, these will reveal differences that are only because of the implementation.

The NASA dataset is used here because it reveals overhead much better than spaces with more expensive metrics. It is known that SimiLite will need more time, so any increase in time spent on metric calculations should only be of benefit. To verify that the runs were not dependent on order, a run on a different randomized order of NASA was also made, and the results were the same as those shown here.

6.1 Index Construction

Compared to MSL the overhead of constructing the indices will be significant. One reason is because for the test-sets used, the MSL indices uses the input file as part of the index, and only has to construct the actual index structures. SimiLite indices, on the other hand, need to insert every element one by one, adding a large constant factor and poor cache utilization. Additionally the automatic index on *rowid* will mean a B-tree will have to be constructed on the table. As demonstrated in figure 6.1, the cost of inserting objects into the main table

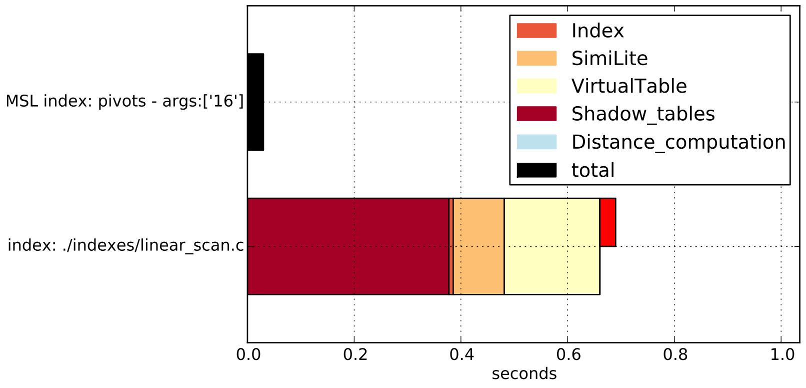


Figure 6.1: Comparison of MSL and SimiLite lower bound construction times.

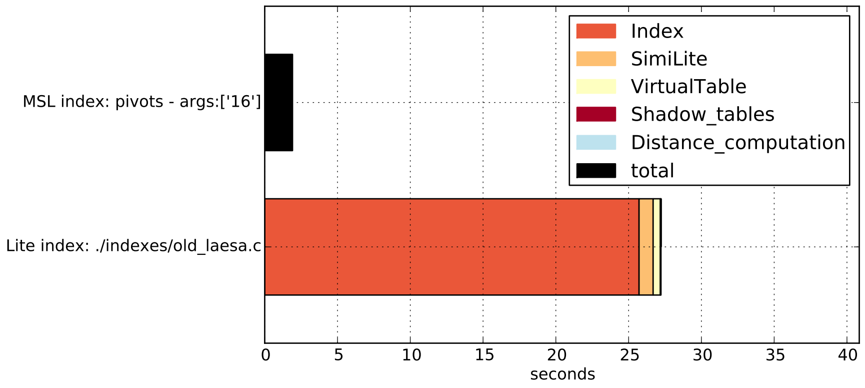


Figure 6.2: Comparison of MSL and the LAESA implementation at the beginning of the project with 500 small queries. Note that this is using some improvements to the SimiLite core.

alone is much higher than for the MSL-indices, and this will is the lower bound for any SimiLite index.

The construction cost, however, is often ignored in the literature since it will be amortized over the queries performed afterwards.

6.2 Querying

The results at the start of the project did not look too positive, (see figure 6.2) with times around 30 times slower than the reference implementation. It was important to uncover whether this was fixable or a huge flaw. This made it important to find what the true overhead

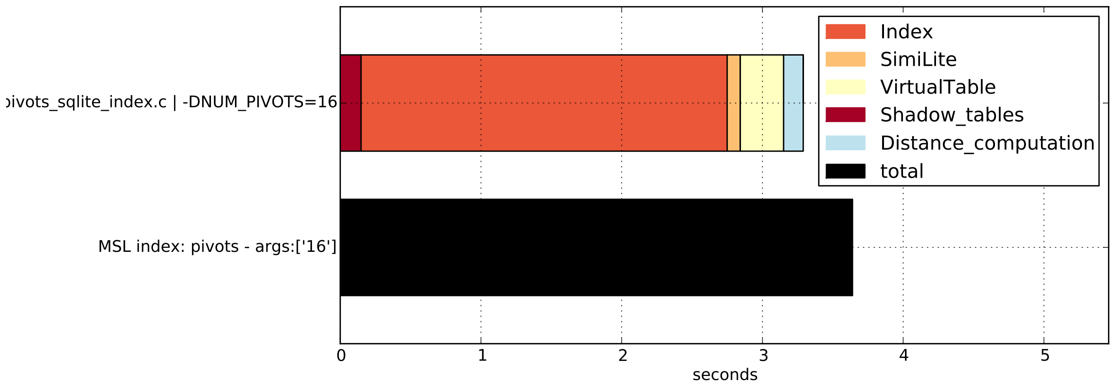


Figure 6.3: Comparison of MSL and SimiLite pivots implementation with 1000 small queries.

of using SQLite was. To accurately measure this an implementation as identical as possible to the MSL version had to be constructed, meaning to keep all the data in ram instead of querying a table. The index was made by using similar memory structures to pivots, and store them as blobs in RAM. The pivot-filtering loop, which iterates through the objects and checks if the object is within bounds, covers over 90% of the runtime and was made identical.

As can be seen in figure 6.3 the difference between the two is quite small, proving that the cost of using the virtual table facility of SQLite can mostly be ignored when analyzing performance. This shows that it is the implementation of the index which is important, which is also the most preferable result.

Once this was known, an attempt was made to create a different index and see if it compared more favorable than LAESA. The choice, SSSTree, will, as seen in figure 6.4, compare well with pivots, but not mvp and bkt, which are also tree-based structures. The results even out a bit for the large queries (figure 6.5). Here the three MSL indices are pretty equivalent, while SSSTree is around 4 times slower.

The final times are displayed in figures 6.6 and 6.7 for small and large queries respectively. The construction times are in figure 6.8. The query-time slowdown is between 5 and 10 times for LAESA depending on the number of results. SSSTree is almost comparable to pivots for the small queries, but has a comparatively larger disadvantage for the large queries.

The speedup compared to a linear scan in SQLite is demonstrated in figures 6.9 and 6.10. For the small queries LAESA is more than

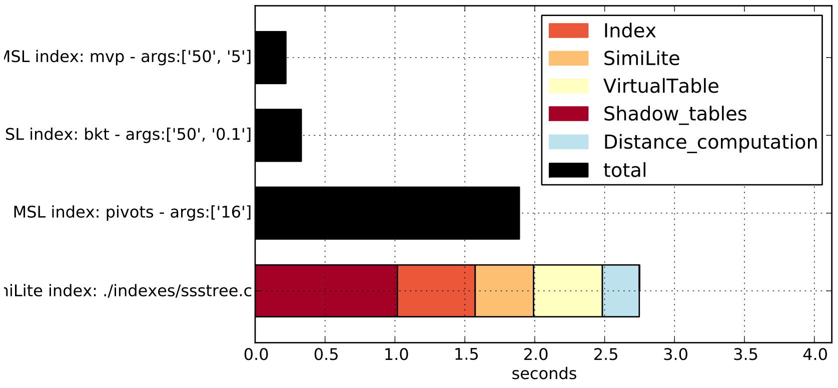


Figure 6.4: Comparison of MSL and SimiLite SSSTree implementation with 500 small queries.

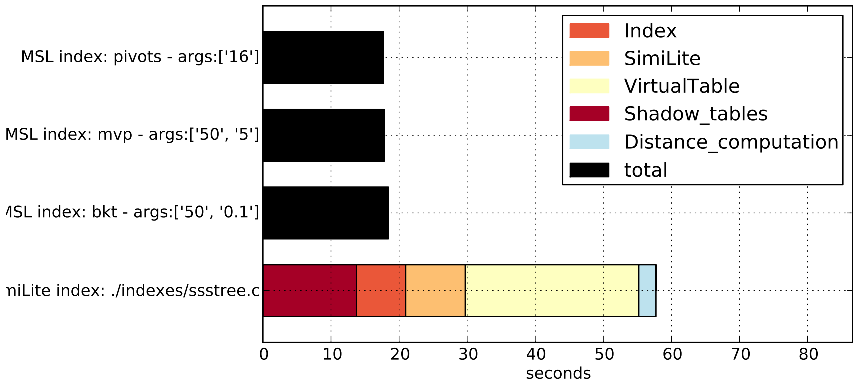


Figure 6.5: Comparison of MSL and SimiLite SSSTree implementation with 500 large queries.

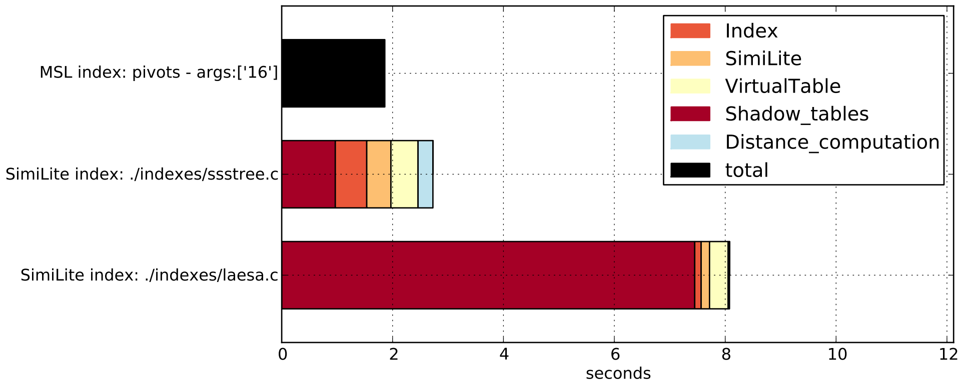


Figure 6.6: Comparison of MSL pivots and final SimiLite LAESA and SSSTree implementations with 500 small range queries.

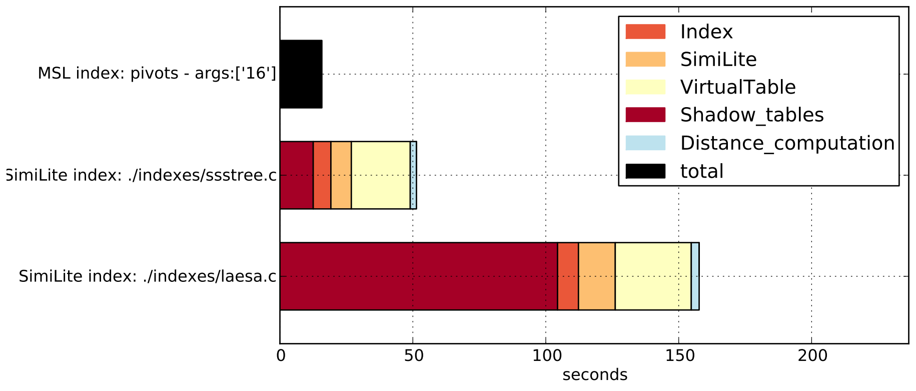


Figure 6.7: Comparison of MSL pivots and final SimiLite LAESA and SSSTree implementations with 500 large range queries.

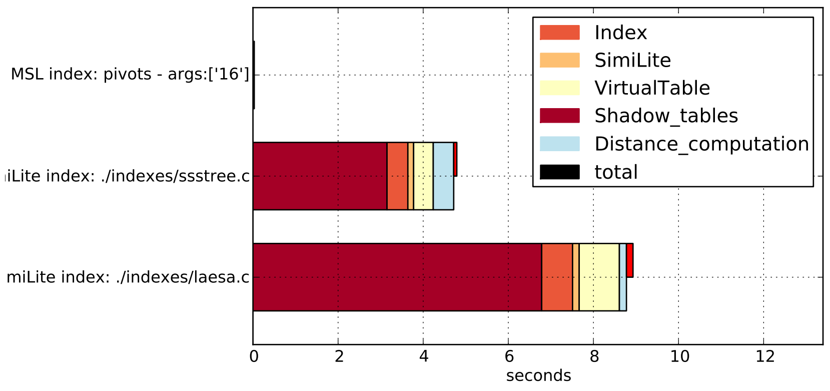


Figure 6.8: Comparison of the construction times of MSL pivots and final SimiLite LAESA and SSSTree implementations.

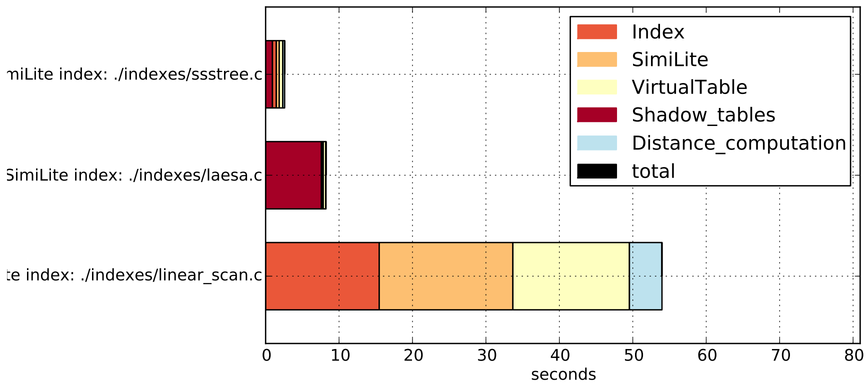


Figure 6.9: Comparison of the times for 500 small range queries on SimiLite LAESA, SSSTree and linear scan implementations.

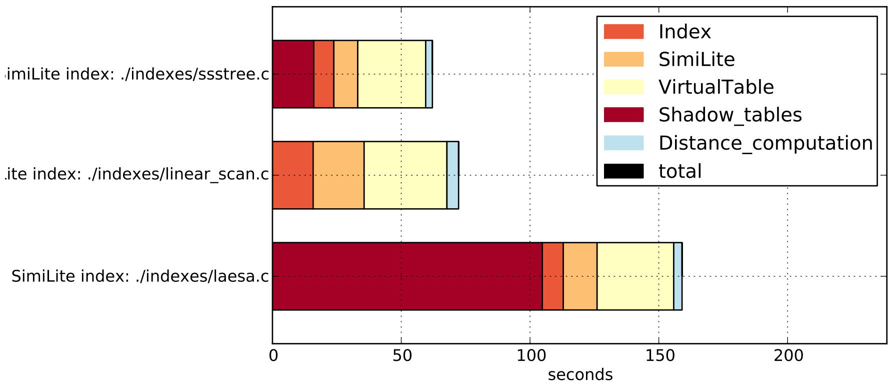


Figure 6.10: Comparison of the times for 500 large range queries on SimiLite LAESA, SSSTree and linear scan implementations.

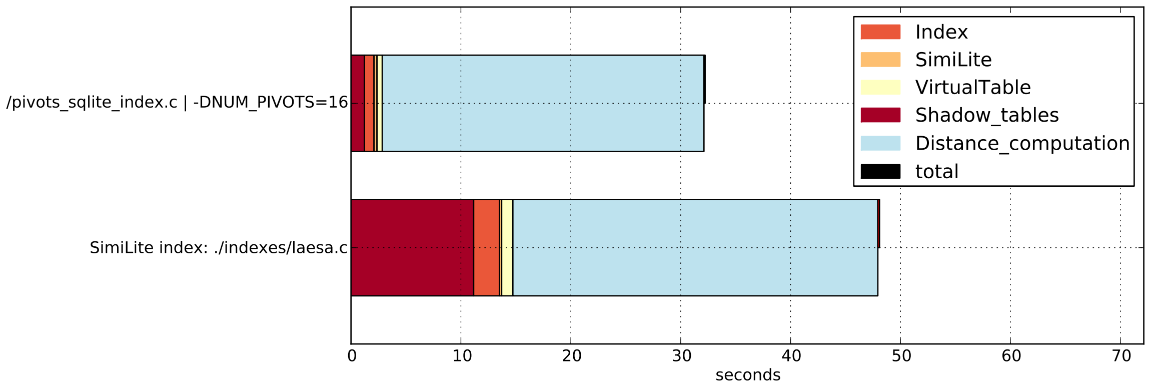


Figure 6.11: Comparison of construction time of LAESA and SimiLite pivots with a metric spending 1 µs sleeping.

5 times faster. The effect is lost for LAESA with the large queries (SSSTree is still a bit faster), but this is near the point where the advantage of indices is lost anyway. Also to be noted is the fact that linear scan uses more distance computations, and SimiLite will take more advantage with the more expensive metrics.

6.3 Effect of varying metric cost

It was stated earlier that the NASA dataset is very good for seeing the overhead, and that a more expensive metric would give more favorable results. As can be seen in figures 6.11, 6.12, 6.13 and 6.14 the times of the indices approach each other when the distance function takes longer. Int these examples it had an added sleep of 1 or 2 µs to simulate

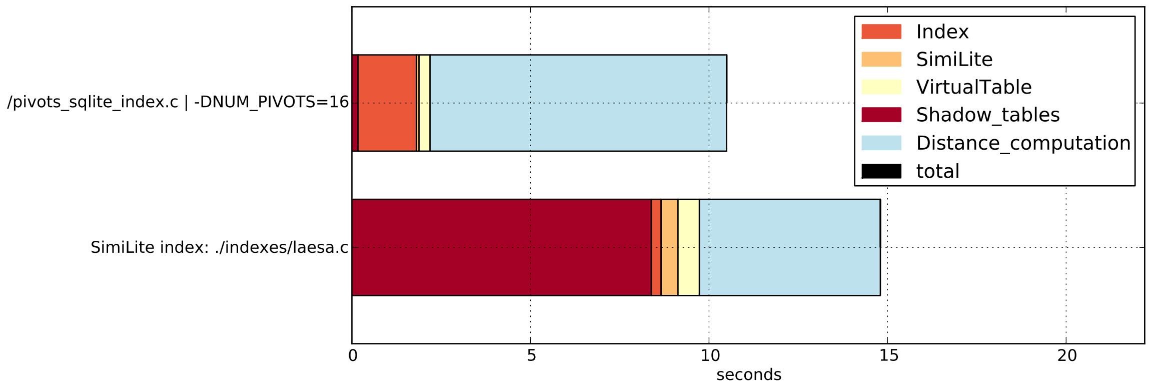


Figure 6.12: Comparison of query time for 500 small range queries with LAESA and SimiLite pivots with a metric spending 1 µs sleeping.

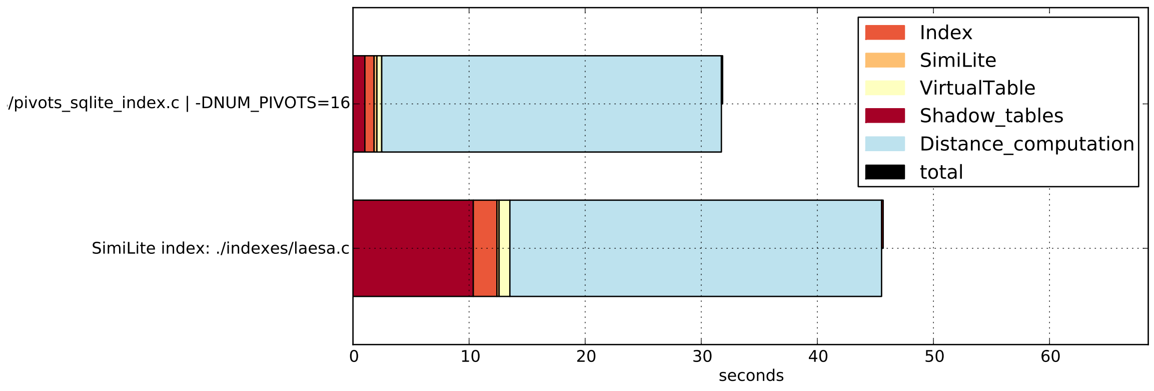


Figure 6.13: Comparison of construction time of LAESA and SimiLite pivots with a metric spending 2 µs sleeping.

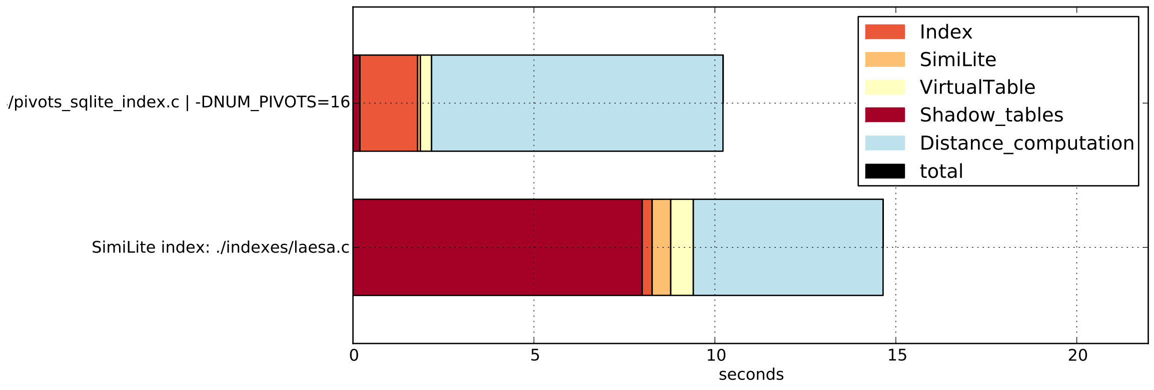


Figure 6.14: Comparison of query time for 500 small range queries with LAESA and SimiLite pivots with a metric spending 2 µs sleeping.

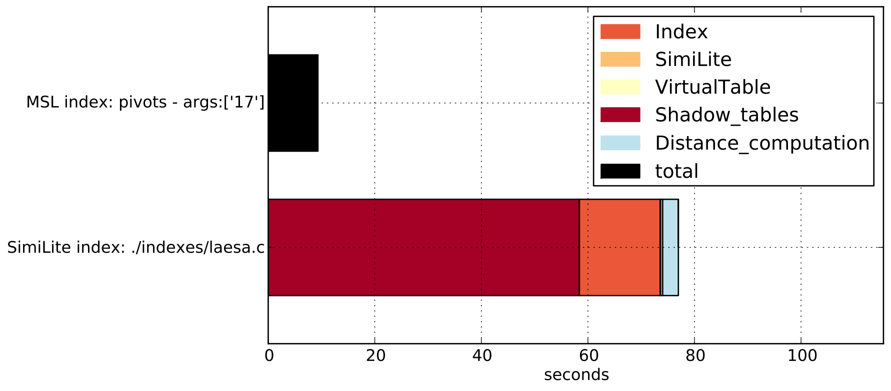


Figure 6.15: Comparison of query time for 500 small kNN queries with LAESA and pivots.

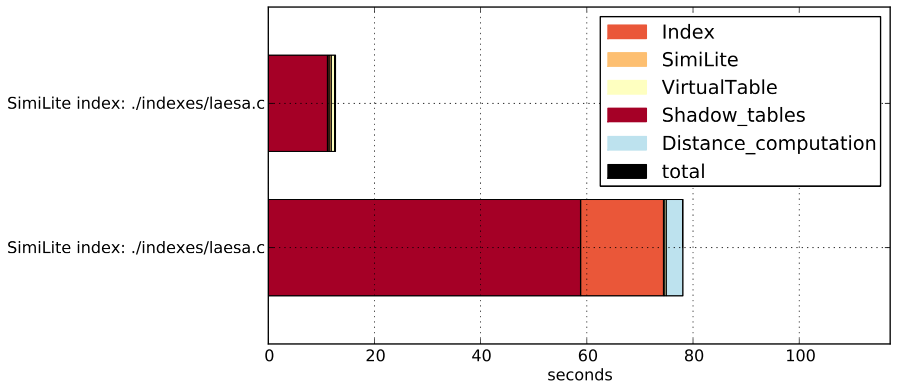


Figure 6.16: Comparison of query time for 500 small kNN queries and range queries with LAESA.

a heavy metric. In fact, since LAESA uses better pivot selection, it will eventually overtake pivots, as can be seen by LAESA having lower distance-computation times in the figures.

In this example the comparison was made with the SimiLite version of pivots to make sure the same distance function was used, and to better show that it was the time spent on distance computations that increased.

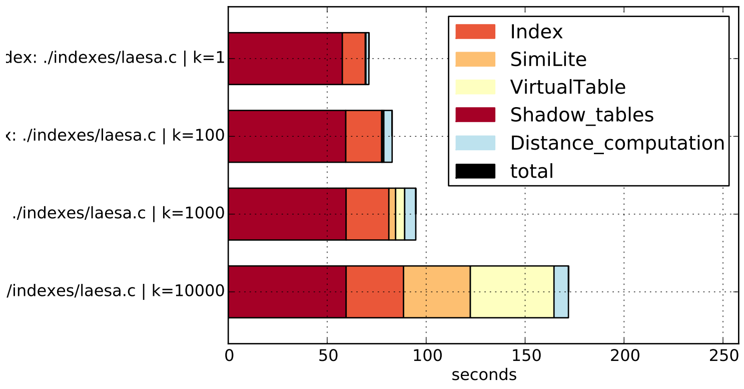


Figure 6.17: Comparison of query time for 500 kNN queries with a k of 1, 100 and 10000.

6.4 kNN queries

LAESA

The performance of the kNN query for LAESA is, as can be seen in figure 6.15, much worse than pivots. While range had a slowdown of approximately 5 compared to pivots for small queries, kNN is slowed down by a factor of approximately 9. This is probably because the boundary has to be traversed so many times, meaning poorer cache performance and branchings when returning.

Compared to the equivalent range query as in figure 6.16, it is easy to see that the cost of kNN is much greater with 6-7 times worse performance. When comparing a kNN query with a k of 1, 100, 1000 and 10000 as in figure 6.17, it seems that simply using a kNN query has a certain cost, and increasing the k has less impact.

SSSTree

The performance is better for the SSSTree implementation of kNN. As seen in the results of figures 6.18 and 6.5 it compares well with MSL pivots for small queries and is less than twice as slow as all the MSL indices for the large queries. It loses a lot in the comparison small queries to the tree-based indices, however, and is 5-10 times slower here.

Comparing to the equivalent range query, as in figure 6.20, shows that the SSSTree implementation is 2-3 times slower.

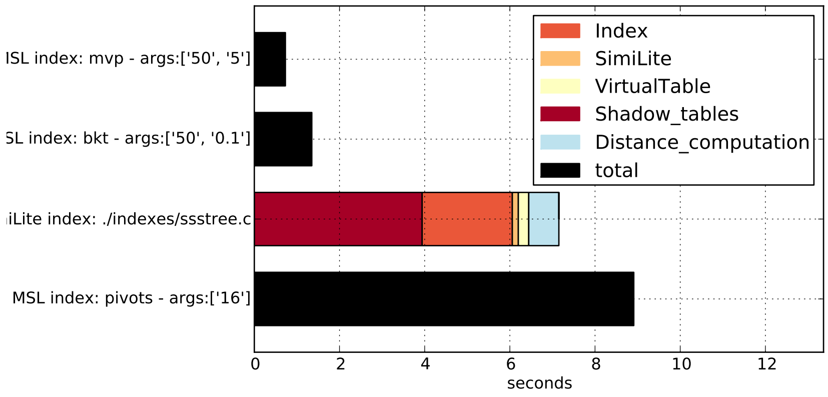


Figure 6.18: Comparison of query time for 500 small kNN queries for MSL pivots, mvp and bkt and SimiLite SSSTree.

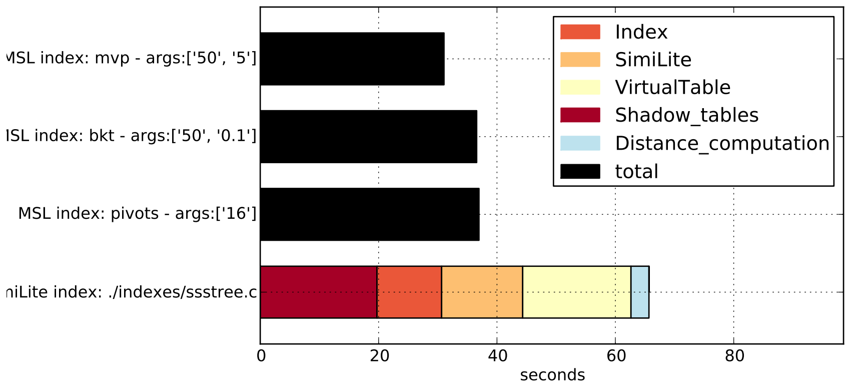


Figure 6.19: Comparison of query time for 500 large kNN queries for MSL pivots, mvp and bkt and SimiLite SSSTree.

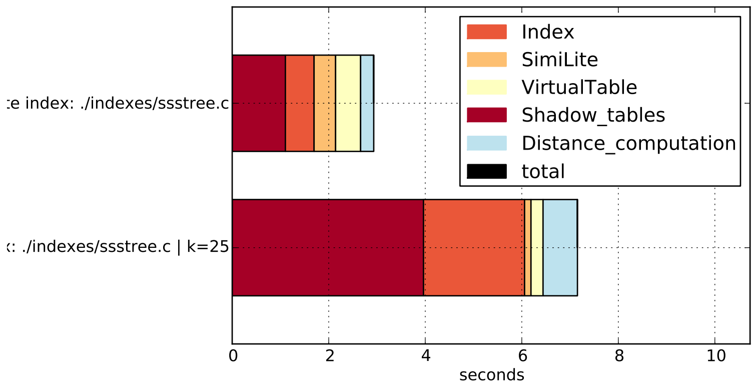


Figure 6.20: Comparison of query time for 500 equivalent kNN and range queries for SimiLite SSSTree.

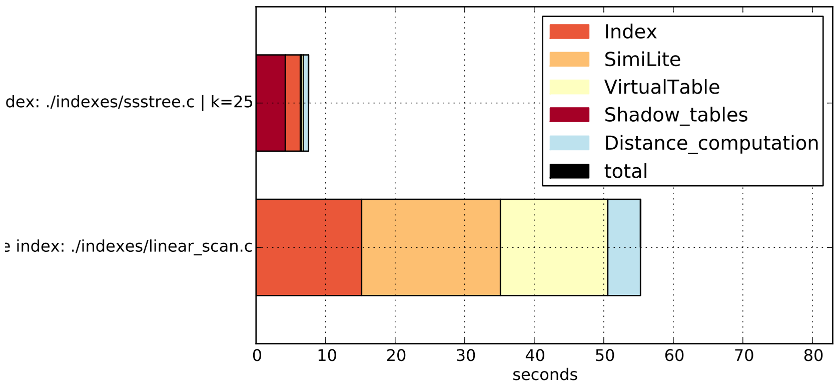


Figure 6.21: Comparison of query time for 500 equivalent SimiLite SSSTree kNN and linear scan range queries.

Chapter 7

Discussion

7.1 Future Work

There are many ways to expand upon SimiLite, two of which are mentioned here. Making SimiLite ready for wider use, and implementing additional indices in SimiLite.

7.1.1 Making SimiLite ready for wider use

The existing implementation has poor error handling, and this must be added to give useful feedback to the user and recover gracefully from errors. More extensive testing should also be done (with edge-cases) to ensure stability.

With regards to syntax; the existing should be carefully reconsidered, and extended to be used as an actual index. How to handle a metric index on more than one column must also be considered.

7.1.2 Implementing additional indices

It would of course be possible to implement many existing indices into SimiLite. One candidate is M-Tree[30], which is optimized with IO-performance in mind.

An other possibility would be to experiment with new indexes with SimiLite. For instance; the idea of a combination of LAESA and SSSTree emerged at the very end of the project. Since it was so late no implementation was done, but since both are implemented already a merging should be possible without too much difficulty. While the chain of elements from root to bucket in SSSTree could potentially be used as pivots (as an extension to the filtering mechanism already used in SSSTree), these are not necessarily far away. Instead a totally external LAESA could be added. While the cost of all operations would potentially "double", the combination of the two filters might be better than either by itself. It could not be worse (with regards to distance computations) than either since the most restrictive of the two schemes would be used in dealing with an object. The pivots of LAESA could additionally be used to filter away some clusters without calculating the distance.

7.2 Conclusion

This project extended the SimiLite implementation with the SSSTree index, and kNN querying capability for both LAESA and SSSTree. The existing LAESA implementation was optimized, and it was learned that changes lessening the load of SQLite had the most impact. Two extensions not mentioned in the paper was added to SSSTree: kNN and pivot filtering, the last of which made the index almost twice as fast. A testing framework was created to make it easier to compare and verify the indices.

In this report it has been shown that SimiLite is 5-10 times slower than a native implementation because of the shadow tables. This is ignoring the cost of the metric however, so the gap will quickly close once they become more expensive. Since the metric index is not meant for cheap metrics anyway, this means that there will be little performance impact in using SimiLite for metric indexing, and for this a user-friendly SQL interface can be used instead. Since SQLite is also widely available through various APIs, it should should be easily possible to add SimiLite to a variety of applications.

While LAESA has fewer distance computations than SSSTree (can be seen in figure 6.6 on page 62 for instance), for cheaper metrics SSSTree is the best choice by far, and can in some cases be directly compared to native implementations. If the alternative is iterating over a table already in an SQLite database, SimiLite can offer 5 times speedup for smaller queries.

The sentiment is that SimiLite should be fully usable as a metric index in the situations where a metric index will be beneficial. If metric indices are to be more widely adopted it must become more accessible, and this is a small step in that direction. Hopefully, it will not be the last.

List of Figures

Bibliography

[1] Maria Stone Jake D. Brutlag, Hilary Hutchinson. User preference and search engine latency. *JSM Proceedings, Qualtiy and Productivity Research Section.*, 2008. [online] http://research.google.com/pubs/archive/34439.pdf [June 7, 2011].

[2] Lokoc J., Hetland M. L., Skopal T., and Christian Beecks. Ptolemaic indexing of the signature quadratic form distance. In *Proc. 4th International Conference on Similarity Search and Applications (SISAP 2011), Lipari, Italy*, 2011.

[3] Magnus Lie Hetland. Ptolemaic indexing. *CoRR*, abs/0911.4384, 2009.

[4] Oracle Spatial and Oracle Locator. [online] http://www.oracle.com/us/products/database/options/spatial/index.htm [October 16, 2010].

[5] Microsoft SQL Server 2008 - Spatial data. [online] http://www.microsoft.com/sqlserver/2008/en/us/spatial-data.aspx [October 16, 2010].

[6] GIS and Spatial Extensions with MySQL. [online] http://dev.mysql.com/tech-resources/articles/4.1/gis-with-mysql.html [October 16, 2010].

[7] PostGIS: Geographic objects for PostgreSQL. [online] http://postgis.refractions.net/ [October 16, 2010].

[8] a complete Spatial DBMS in a nutshell. [online] http://www.gaia-gis.it/spatialite/ [October 16, 2010].

[9] Bin Yao, Feifei Li, and Piyush Kumar. K nearest neighbor queries and knn-joins in large relational databases (almost) for free.

[10] A. Kumaran, P.K. Chowdary, and J.R. Haritsa. On pushing multilingual query operators into relational engines. 2006.

[11] Introduction to GiST. [online] `http://www.sai.msu.su/~megera/postgres/gist/doc/intro.shtml` [October 28, 2010].

[12] Erik Bagge Ottesen. Similarity Search in Large Databases using Metric Indexing and Standard Database Access Methods, 2009.

[13] Rui Mao Daniel Miranker, Weijia Xu. Mobios: a metric-space dbms to support biological discovery. 2003. [online] `http://www.cs.utexas.edu/~mobios/` [October 31, 2010].

[14] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw Hill Higher Education, 2002.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[16] Magnus Lie Hetland. The basic principles of metric indexing. 2009.

[17] Gisli R. Hjaltason and Hanan Samet. Index-driven similarity search in metric spaces (survey article). *ACM Trans. Database Syst.*, 28:517–580, December 2003.

[18] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in metric spaces. *ACM Comput. Surv.*, 33:273–321, September 2001.

[19] María Luisa Micó, José Oncina, and Enrique Vidal. A new version of the nearest-neighbour approximating and eliminating search algorithm (aesa) with linear preprocessing time and memory requirements. *Pattern Recogn. Lett.*, 15:9–17, January 1994.

[20] E V Ruiz. An algorithm for finding nearest neighbours in (approximately) constant average time. *Pattern Recogn. Lett.*, 4:145–157, July 1986.

[21] Enrique Vidal. New formulation and improvements of the nearest-neighbour approximating and eliminating search algorithm (aesa). *Pattern Recognition Letters*, 15(1):1 – 7, 1994.

[22] Nieves Brisaboa, Oscar Pedreira, Diego Seco, Roberto Solar, and Roberto Uribe. Clustering-based similarity search in metric spaces with sparse spatial centers. 4910:186–197, 2008.

[23] Karina Figueroa, Gonzalo Navarro, Edgar Chávez. Metric spaces library, 2007. [online] `http://www.sisap.org/Metric_Space_Library.html` [December 15, 2010].

[24] SQLite homepage. [online] http://www.sqlite.org/ [June 7, 2011].

[25] Mike Owens. *The Definitive Guide to SQLite*. Apress, 2006.

[26] The SQLite R*Tree Module. [online] http://www.sqlite.org/rtree.html [Jun 5, 2011].

[27] SQLite FTS3 and FTS4 Extensions. [online] http://www.sqlite.org/fts3.html [Jun 5, 2011].

[28] Binary heap. [online] http://cprogramminglanguage.net/binary-heap-c-code.aspx [June 10, 2011].

[29] Datatypes In SQLite Version 3. [online] http://www.sqlite.org/datatype3.html [June 11, 2011].

[30] P Ciaccia, M Patella, and P Zezula. M-tree: An efficient access method for similarity search in metric spaces. *VLDB*, Athens, Gr:426–435, 1997.