Evaluating Discrimination for Database Systems

Felix Plymouth

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2017

Abstract

This project aims to explore the application of a discrimination based sorting strategy to database systems. For this purpose a version of SQLite that had a partial implementation of discrimination based sorting was used. This SQLite implementation was improved upon and then benchmarked. The benchmarks used were TPC-C, for which an open source implementation was found, and TPC-H, for which an implementation was created. Finally, to get a clearer understanding of the impact discrimination sort has on sorting performance, a number of simpler tests were performed. It is found that while discrimination based sorting provides an advantage over comparison based sorting for large enough inputs the threshold is high and as such care would have to be taken in the types of systems it is applied to.

Acknowledgments

I wish to thank Prof. Philip Wadler for providing support, input, and feedback during the entire project in his role as supervisor. Without his help none of this would have been possible.

I wish to thank Mr. Ma Hoi Hei for providing the source code for his modified version of SQLite and the source code for the benchmark he developed.

Table of Contents

Chapter 1

Introduction

One of the first things taught to computer science students is that n items can not be sorted in less than $O(n * log(n))$ time. But sorting can be done in linear time, and such algorithms have existed for more than 100 years. The main reason that they are not used more often is that they only apply to specific data types, such as strings and integers. A solution to this issue was presented quite recently in a paper by Prof. Henglein [Henglein, 2012] by the introduction of discriminators, extending this type of sorting algorithm to the vast majority of data types.

At the same time, the need for high performance database systems continues to grow as the web grows and in such systems every bit of performance is important. One of the main tasks large scale database systems are often required to perform is the sorting of data, both as part of queries that explicitly ask for sorted data and as part of queries that include joins between tables. As such much could be gained from faster sorting in database systems.

In this project the applicability of discriminators to database systems in general and SQLite in particular will be evaluated. This builds on work done in [Hoi Hei, 2016] where a discriminator based sorting algorithm was implemented as part of the SQLite database system. The implementation and evaluation was somewhat lacking, and in this project said work will be built on to provide both a more robust system, and a more robust evaluation of said system. However, the results from [Hoi Hei, 2016] were promising, as a consistent decrease in query execution time was observed for several datatypes, including strings of varying lengths, integers, and floating points.

- Chapter 2 will present what a discriminator is (2.1), some background on SQLite and how the discriminator for SQLite records is implemented (2.2), and how databases are generally evaluated (2.3).
- Chapter 3 will quickly review some of the work done for this project that did not in the end contribute to the final results.
- In Chapter 4 the results of evaluating the implementation using TPC-C, an online transaction processing benchmark, will be presented.
- In Chapter 5 the results of evaluating the implementation using TPC-H, a deci-

sion support benchmark, will be presented.

- In Chapter 6 the findings in [Hoi Hei, 2016] will be re-evaluated.

The main pieces of work done for this project were:

- Background reading and research.
- Understanding, evaluating, and building on the SQLite implementation of Discrimination sort that was developed in [Hoi Hei, 2016].
- Finding and to some extent repairing a TPC-C benchmarking program for SQLite.
- Constructing a TPC-H implementation for SQLite.
- Benchmarking the final implementation and evaluating the results of said benchmarks.

This report assumes a basic understanding of databases, no more than what might be gained from an introductory course on the subject, along with the ability to understand basic Haskell and C code.

Chapter 2

Background

2.1 Discrimination Sort

The concept for discrimination sort is introduced in [Henglein, 2012] as a part of introducing the concept of discriminators. A discriminator is, in it's most basic form, a function that takes a list of key value pairs and returns a list of lists of values sorted and partitioned by their keys. An example; for a discriminator on strings called `stringDisc`.

- `stringDisc [("c",1),("b",2),("a",3)] = [[3],[2],[1]]`
- `stringDisc [("c",1),("b",2),("c",3)] = [[2],[1,3]]`
- `stringDisc [("c",1),("b",2),("a",2)] = [[3],[2],[2]]`

In truth, the values do not have to be sorted by their keys, and if they are it's called an order discriminator. However, since discriminators will only be used for sorting in this report all discriminators are assumed to be order discriminators.

The interesting point is that it is possible to construct a discriminator that has worst case $O(n)$ execution time, where n is the list length, for all regular recursive first-order types. This is achieved through recursive application of a distribution sorting algorithm. These are best explained by example:

1. Start with a list of values to be sorted: $[11,1,12,21,3,2,10]$
2. Distribute each value into a bucket indexed on the first part of the value, so 1 for 10, 2 for 21, and 0 for 1: $[[1,3,2],[11,12,10],[21]]$
3. Apply the above step recursively on each bucket until there are no parts left to distribute on, in this case only one step is required: $[[[1],[2],[3]],[[10],[11],[12]],[[21]]]$
4. Flatten back into a list again: $[1,2,3,10,11,12,21]$

Regular recursive first-order types, also defined in [Henglein, 2012], is the set of types that can be defined using the following constructs: the unit type, product types

(pairs), sum types (either or), type variables and μ-abstraction (for recursive definition, this is the term used in [Henglein, 2012] and it behaves like a fixpoint for types), and finally the integers. Also, elements have to be finite, but since most computers are finite this should be fine. Examples of these kind of types are floating points, strings, trees, and lists. Which includes most things that might be sorted in a database. Elements of these types are any finite element generated by the grammar, $v ::= c|()|inl\ v|inr\ v|(v,v)|fold(v)$ where c ranges over the integers.

Finally, to understand discriminators a concept of ordering relations are needed, as the implementation of discriminators mirror them. An ordering relation R over the type T is a relation, which is defined as a set of tuples $R \subseteq (T \times T)$. Then for $t_1, t_2 \in T$ if $(t_1, t_2) \in R$ then $t_1 \leq_R t_2$. Consequently if $(t_1, t_2) \in R \wedge (t_2, t_1) \notin R$ then $t_1 <_R t_2$ and $(t_2, t_1) \in R \wedge (t_1, t_2) \in R$ then $t_1 \equiv_R t_2$. Essentially, it defines an ordering of the elements of some type.

These are the basic ordering relations and how they are implemented as discriminators in Haskell, along with the implementation the asymptotic run time will also be provided. Please note that the version of these functions presented here are not the same as the ones presented in [Henglein, 2012]. They have been rewritten since the originals don't lend themselves to easy understanding as they are built on top of an additional layer of abstraction that is not needed for this project.

- The empty relation over any set S, where no elements are related. Therefor all keys are unique.

  ```
  empty :: [(k,a)]-> [[a]]
  empty xs = [[v]|(k,v) <- xs]
  ```

  Run time: $O(n)$

- The trivial relation over any set S , where all elements are related. Therefor all keys are equal.

  ```
  trivial :: [(k,a)]-> [[a]]
  trivial xs = [[v|(k,v) <- xs]]
  ```

  Run time: $O(n)$.

- And finally, for non-negative n, the standard order $[n] = \{(k,l)|0 \leq k \leq l \leq n\}$ for any set S such that $\{0, 1, ..., n-1, n\} \subseteq S \subseteq \mathbb{Z}$

  ```
  standard :: Int -> [(Int,a)] -> [[a]]
  standard n xs =
          filter (not . null) (map reverse (elems
                  (accumArray update [] (0, n-1) xs)))
          where update vs v = v:vs
  ```

  This uses `accumArray`, a Haskell inbuilt function that is used to build an array. It takes a range of indexes `(0,n-1)` and initializes each index in that range to a provided default `[]`. Then for each key value pair (k,v), v is inserted into the array at index k by replacing what was at that index, call this vs, with `update vs v`.

To read every index of an array back into a list `elems` is used.

The sub array is then reversed, so that the sort remains stable, this is necessary since elements are prepended to the elements of the array, not appended. A sort function is considered stable if and only if for all pairs of elements a,b in the input if $a = b$ and a appears before b in the input then a appears before b in the output.

Finally, indexes that do not contain any elements are removed from the list.

This might seems somewhat needlessly complex but it has the important property that it runs in $O(n)$ where n is the length of xs. The same performance could not have been achieved using Haskell lists.

Note that the implemented discriminators all have roughly the same type signature, so to simplify the type signature a new type definition is created for discriminators.

```
type Disc k = forall v. [(k,v)] -> [[v]]
```

This set can then be extended using a number of ordering constructs that map onto the allowed type constructs for Regular recursive first-order types. An ordering construct can be viewed as a function that takes one or more ordering relations, and potentially some other argument, and returns an ordering relation. The full set of ordering relations presented in Henglein [2012] is not needed to understand this report so an abridged set is presented here:

Given ordering relations R_1, R_2 over types T_1, T_2 respectively:

- The left first sum order $R_1 + R_2$ over $T_1 + T_2$, as usual $inl\, x < inr\, y$, $inl\, x \leq inl\, y \iff x \leq_{R_1} y$, and $inr\, x \leq inr\, y \iff x \leq_{R_2} y$

  ```
  sumOrd :: Disc a -> Disc b -> Disc (Either a  b)
  sumOrd dleft dright xs = (dleft (lefts elems)) ++ (dright (rights elems))
          where
                  change (Left k,v) = Left (k,v)
                  change (Right k,v) = Right (k,v)
                  elems  = map change xs
  ```

  Note that *Either* is the built in Haskell representation for sum types. Run time: $O(dleft(n) + dright(n) + n)$: Cost of processing the lefts plus cost of processing rights plus overhead from filtering and concatenation.

- Lexicographic product order $R_1 \times R_2$ over $T_1 \times T_2$ $(x_1, x_2) \leq (y_1, y_2) \iff x_1 <_{R_1} y_1 \vee (x_1 \equiv_{R_1} y_1 \wedge x_2 \leq_{R_2} y_2)$

  ```
  prodOrd :: Disc a -> Disc b -> Disc (a,b)
  prodOrd dleft dright xs =concat (map dright (dleft elems))
          where
                  change ((k1,k2),v) = (k1,(k2,v))
                  elems  =  map change xs
  ```

  Run time: $O(dleft(n) + dright(n) + n)$, cost of processing element one plus cost

of processing element two plus overheads from pre-processing and concatenation.

- Preimage ordering $f^{-1}(R_2)$ under f over T_1, $x \leq y \iff f(x) \leq_{R_2} f(y)$, where f is a function $T_1 \to T_2$.

```
preOrd :: (a -> b) -> Disc b -> Disc a
preOrd f disc xs =disc elems
        where
                change (k,v) = (f k,v)
                elems  =  map change xs
```

  Runtime: $O(n * f(1) + disc(n))$ cost of transformation plus cost of discriminator.

- Lexicographic list ordering over T_1^* (List with elements of type T_1), where $x = [x_1, ..., x_m] \leq y = [y_1, ..., y_n] \iff \exists i \leq m+1.(i = m+1) \vee x_1 <_{R_1} yi) \wedge \forall j < i.x_j \equiv_{R_1} y_j$. Also known as the standard ordering on lists.

```
listOrd :: Disc a -> Disc [a]
listOrd disc xs
        | not (null nulls) = nulls:(concat (map (listOrd disc) (disc elems)))
        | otherwise = (concat (map (listOrd disc) (disc elems)))
        where
                change (k:ks,v) = (k,(ks,v))
                nulls = map snd (filter (null . fst) xs)
                elems  =  map change (filter (not . null . fst) xs)
```

  Run time: $O(disc(n) * k + k * n)$ where k is the maximum length of any key in xs. Derived as follows, at each level of recursion **disc** is run at most once per element, and filtering and concatenation is done at most once per element too. The maximum depth of recursion is the same as k since the function stops when each key is empty and the length of each key is always reduced by one in each iteration.

It should at this point be obvious that as long as a discriminator is constructed from just these elements the asymptotic run time will never exceed $O(n)$. However, in reality the run time does scale with the complexity of each element. Sorting length 100 strings take 10 times longer than sorting length 10 strings. This is unusual for sorting algorithms for which comparison is generally assumed to be constant time in regards to the length of the keys. This problem can to some extent be alleviated by adding early return to our discriminators, which is easily accomplished by adding the following line to each function given above.

```
ord [(_,v)] = [[v]]
```

Where ord is a stand in for the name of the function. Essentially, input of length one is simply returned without further modification. For an example of why this is significant, consider sorting the single character strings `["a",...,"z"]` and the same strings but with 100 random characters appended to each string. Call these inputs *short* and *long* respectively. If the above change is not made to the discriminators, sorting *long* will take 100 times longer to sort than *short*. However, with the modified version, it only

takes one character to determine where each string places in the ordering of the input, as such it will take the same amount of time to sort *long* as it does to sort *short*.

Essentially, the unmodified version scales with the complexity of the keys, while the modified version scales with the length of the input that is required to determine each elements place in the sorted output. In other words, for n well distributed inputs of length l, the unmodified version has an average execution time that scales with $l * n$ while the modified version has an average case execution time that scales with $n * min(l, log(n))$. They have the same asymptotic run time, since $log(n)$ will be greater than l for a high enough n, but the latter is a significant improvement for small values of n.

To make it clearer how these are used here are a couple of examples. Strings (which are lists of characters in Haskell), can be given as $stringToBytes^{-1}([255]^*)$ where $stringToBytes$ is a function that maps a String to it's byte representation. Remember that $[n]$ denotes the standard ordering for integers up to n. Implemented as:

```
stringOrd :: Disc [Char]
stringOrd = preOrd stringToBytes (listOrd (standard 256))
        where
                stringToBytes = map ord
```

Another rather important example is the different ways in which and ordering for unsigned integers can be constructed, 16 bit ones make for a good example. The trivial representation is simply $[2^{16} - 1]$, but this has the problem of not being extensible to other integer sizes due to memory problems. Remember that `standard` allocates an array of the length provided, as such as the integer size grows so does the array, this is fine for 16 bit integers but sorting 32 bit integers would require an array of length 2^{32} which amounts to around 4 GB of space and 64 bit integers would require 18 EB (exa bytes).

A different, somewhat more clever way is the following. $unit16ToBytes^{-1}([255] \times [255])$ where $unint16ToBytes(x) = (x >> 8\&255, x\&255)$ (places the top 8 bytes in the first element of a tuple and the lower 8 bytes in the second). It should be evident that this produces the same ordering as $[2^{16} - 1]$, but it can easily be extended to larger integers, for example for a 32 bit integer a function that maps elements to an 4-tuple of bytes can be used and the ordering relation becomes $unit32ToBytes^{-1}([255] \times [255] \times [255] \times [255])$. The 16 bit version can be implemented as:

```
uint16Ord :: Disc Int
uint16Ord xs = preOrd uint16ToBytes (prodOrd (standard 0xFF) (standard 0xFF)) x
        where
                uint16ToBytes x = (mod (shiftL x 8) 0xFF, mod x 0xFF)
```

2.2 A discriminator for SQLite

With this knowledge of discriminators, it is possible to apply it to a database system, in [Hoi Hei, 2016] SQLite was used so it will also be used here. There are a few ways

in which SQLite differs from other major database systems which makes it preferable.

- It's open source, in the sense that the source code is available and free to use and modify [SQLite, a]. It's a rather obvious criteria but it is worth mentioning. It should however be noted that SQLite isn't particularly open to contribution, since it is built and maintained by a small company rather than the community, which is often the case with other open source projects.
- It's thoroughly documented, both through a comprehensive wiki and through extensive code comments. This should make modifying it much easier, though due to the size of the code base this is not always so simple.
- It is easy to deploy. Most database systems are set up to run in client/server mode and they are meant to be online permanently. This means that they generally have a rather complex setup, installation, and startup process, which has to be repeated every time a new modified version is created.SQLite is instead meant to be deployed as an embedded database system, meaning that it is compiled into the application as a library and it's database files are stored on the client as a normal file. This creates a much simpler deployment pipeline whenever a new version is produced.
- Finally, it is widely used. In fact it is probably the most widely deployed database system that exists [SQLite, f]. For example, it is deployed with every single android device [SQLite, b]. This is important since, if the performance of SQLite can be improved, then the performance of a large number of applications can be improved.

First some background on how SQLite implements sorting and the data types involved.

2.2.1 SQLite background

SQLites main sorting routine, the one used to sort database records, is implemented in *vdbe_sort.c* [SQLite, e], in fact, two sorting algorithms are implemented, one internal and one external. An external sorting algorithm is one that stores some of the data on disk while the rest is being sorted, as opposed to an internal sorting algorithm that keeps all records to be sorted in memory. The former generally uses the latter as a subroutine. External sorting algorithms will be discussed further in 3.1.

The datatype sorted on is a list SQLite records [SQLite, e], a format for representing rows of a table, the details of which can be found in the SQLite documentation [SQLite, d]. Each record consists of a header, which contains the size of the header, and the datatype of each cell in the record, given as an integer, followed by a body which contains the actual data of the row. The datatypes are as follows [SQLite, c]:

- **Null**: the **Null** datatype has only a single value *Null* and in the header it is represented as the value 0, it does not take up any space in the body.
- **Integer**: the **Integer** datatype is used to store signed integers in a compact format. In the header these are represented as the numbers 1 through 6 inclusive,

plus 8 and 9. The number 1,2,3,4,5, and 6 signify a 8,16,24,32,48, and 64 bit twos-complement integer respectively and they consume that number of bits in body. 8 and 9 represent the values 0 and 1 respectively, and consume no space in the body. When picking a format to store an Integer, the smallest format that can represent the Integer without loss is used.

- **Real**: the **Real** datatype is used to store big-endian IEEE 754-2008 64-bit floating point numbers. In the header it is represented by the number 7.
- **String**: the **String** datatype is used to represent strings stored in some format, the exact format is specified in the preamble of the database file, and can be any one of *UTF-8*, *UTF-16le*, and *UTF-16be*, *UTF-8* is the default. In the header strings are represented by any odd number N above 13, and the string has length $(N-13)/2$. Strings are not stored in a Null terminated format.
- **Blob**: The **Blob** datatype is used to store arbitrary length binary data. In the header blobs are represented as any even number N above 12, and the data contains $(N-12)/2$ bytes.

SQLite defines the default ordering of these to be, in order from smallest to largest:

1. **Null**
2. Numerics, a merger of **Integer** and **Real**, internally ordered in the standard numeric order
3. **String**, internally ordered in byte order, which is the same as alphabetic order for *UTF-8* and *UTF-16le*.
4. **Blob**, internally ordered in byte order.

There are no guarantees that the same column of two records, database rows, from the same table will contain a value of the same type. This is somewhat unusual for SQL database systems but it is not excluded by the SQL standard.

It is also possible that a specific column of the table that is being sorted should be ordered in reverse order, this is passed in as a parameter to the sorting function. However discriminators do not accept meta parameters, instead something like this should be built into the discriminator itself. As such it will be left out in our discussion in section 2.2.2 however it will be discussed in section 2.2.3.

2.2.2 A discriminator for the SQLite records

The first step towards a discriminator for SQLite records is to build a discriminator for each of SQLites datatypes. for **Null** the discriminator is simply the `trivial` discriminator.

String and **Blob** share the same discriminator, which is `byteOrd = listOrd (standard 256)`, also known as a standard byte ordering.

However, the ordering for numerics is somewhat more complex since they are not easily represented as a single data type, which is needed to be able to construct a discriminator. A data format which can without loss represent all 64 bit integers and all IEEE 754-2008 64-bit floating point numbers is needed. From now on IEEE 754-2008 64-bit floating points will be referred to as 64-bit floats. Fortunately, such a format already exists in the x86 extended precision format, from here on referred to as a 80-bit float. This format has a sign bit s, a 15 bit exponent e, an single bit integer part m_1 and a 63 bit fraction m_2, the integer part and the fraction can be combined into a single value $m = m_1.m_2$. The value of an 80 bit float is computed as $(-1)^s * 2^{(e-16383)} * m$. See Figure 2.1 for how this is laid out in memory. The integer part is in almost all relevant cases 1, the one exception being the number 0.

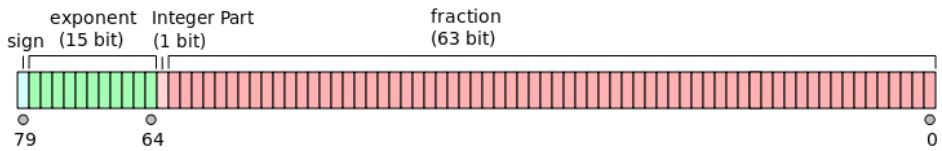

Figure 2.1: 80 bit floating point format

Once both datatypes can be converted to this format a strategy for constructing a discriminator for them is needed, or essentially, how can the 80-bit float format be decomposed into a sequence of unsigned integers that can then be sorted on.

First note that the value taken as it is and written into a sequence of 10 bytes, high byte first, can be discriminated on to produce a correct ordering for all positive numbers. However, there are two problems when the negative numbers are included.

1. Since the first bit of all negative 80 bit floats is 1, the negative numbers appear to be larger than the positive numbers. This is easily fixed by flipping the first bit.

2. The negative numbers come out in reverse order, this can be fixed by flipping all bits in the number if it is negative.

To transform 64 bit floats or integers into 80 bit floats built in hardware can be used or a method can be implemented manually, which will be discussed in 3.2.1.

The discriminator for numerics can then be represented as follows, with map_float80 performing the transformation discussed above:

```
numericOrd :: Disk (Float80)
numericOrd = preOrd map_float80 (listOrd (standard 256))
```

The smaller discriminators can then be combined into one larger discriminator.

```
recordOrd :: Disk [Either Null (Either Float80 (Either String Blob))]
numericOrd = listOrd (sumOrd trivial (sumOrd numericOrd (sumOrd byteOrd byteOrd)))
```

Remember that *byteOrd* is the ordering for **String** and **Blob**, and *trivial* is the ordering for **Null**.

2.2.3 Translating to C, and the real world

So far discriminators have been presented in the form of Haskell code, and for a very good reason, they are very much a functional programming concept and as such it is easier to teach them in terms of a functional language. But SQLite is written in C so they have to be translated. The main goal is the supposed performance improvements that discrimination sort has over comparative sorting and as such parts of the Haskell implementation can be left out. Furthermore, there are a few limitations, the primary one being that the implementation can not use any additional memory apart from the stack due to the fact that the discriminator will be operating in a memory critical environment, especially when it is invoked as part of an external sort.

The function that needs to be changed has the following signature:

```
static int vdbeSorterSort(SortSubtask *pTask, SorterList *pList);
```

The first argument `SortSubtask *pTask` contains meta information about the sort to be carried out, such as information about the character encoding and which fields should be reversed. The second argument `SorterList *pList` is a container for an unsorted linked list of SQLite records, the task of the function is to sort this linked list. Each record is tied to a row of a SQLite table, but only the cells of the row that are to be sorted on are passed to the function. SQLite uses other functionality to track which record belongs to which row. The return value of the function is simply an error code to be used for things like out of memory errors.

As such the following changes were made to discriminators for this implementation to be feasible.

- The framework for generic discriminators outlined in section 2.1 can be discarded. Instead a single discriminator for the SQLite record format is implemented. This makes the implementation work easier and should also improve performance.
- Instead of sorting values by associated keys only keys need to be sorted. As mentioned only the parts of each row that are to be sorted on are passed to this function, these can be thought of as the keys, while the database rows they are tied to can be thought of as the values.
- Instead of returning a list of lists, to keep track of which elements are equal, a sorted list is returned which is simply the flattened version of the list of lists that a discriminator would have produced.
- The entire key along with some supporting information is passed to the sub discriminators instead of decomposing keys into parts to be passed to each sub discriminator. Supporting information includes which cell is currently being sorted on and the position within that cell. This saves a significant amount of memory that would have otherwise been consumed by the decomposed keys.

The architecture of the final system is illustrated in Figure 2.2 with each discriminator going through the following steps in order:

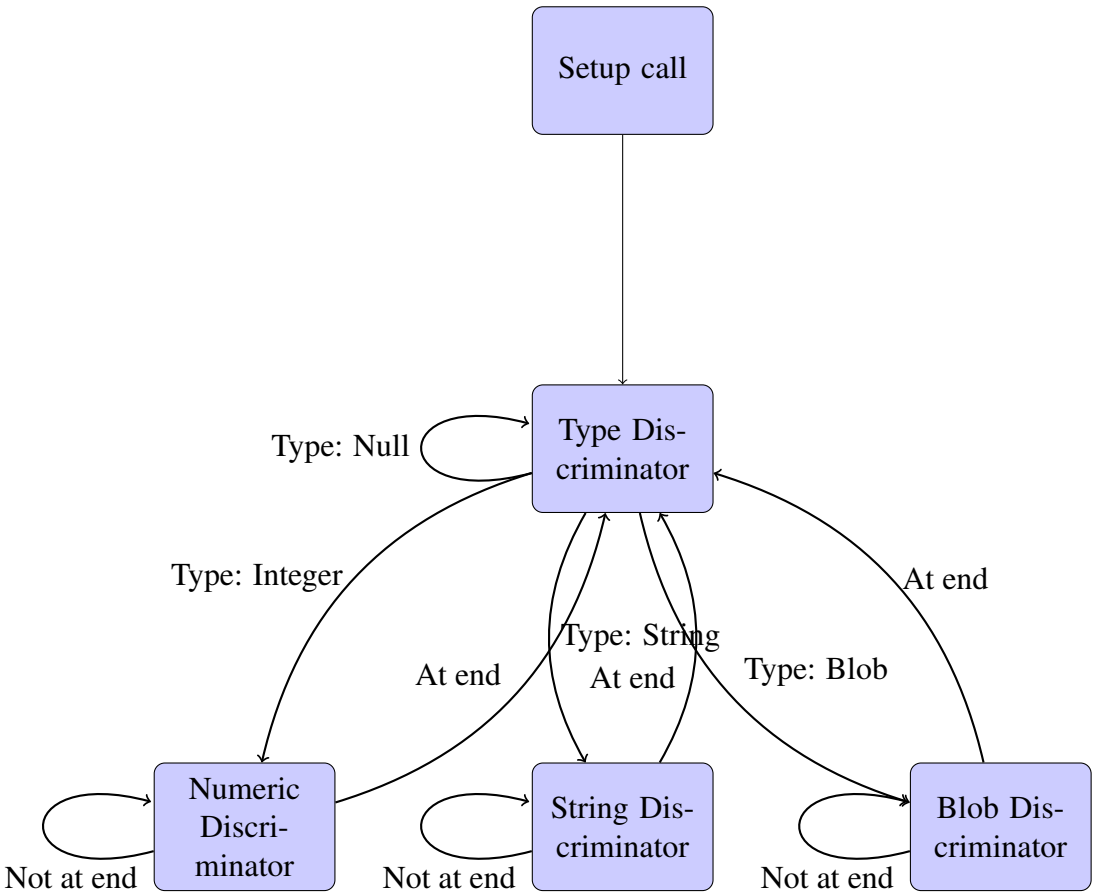

Figure 2.2: Discriminator architecture

1. Distribute each element to a bucket, the Type discriminator use 4 buckets, one for each type, while the others use 257, one for each byte value and one for cell end.

2. Call the next discriminator on each bucket, the next discriminator being:

   - The same discriminator, if the cell contains more bytes or if it's the Null bucket in the Type discriminator.

   - The Type discriminator, if the end of a value in a non Type discriminator has been reached.

   - A Non Type Discriminator appropriate for the bucket in the Type Discriminator.

3. Gather the now sorted elements in each sub bucket, in order, into the argument bucket of the function, here the buckets are traversed in reverse order if the column that is being sorted on is to be sorted in reverse order.

As an example, to sort a linked list of SQLite records representing the following tuples $(Null, "b"), (Null, "a")$ the following steps are performed. Each step inwards represents a recursive call.

1. The setup call extracts the linked list and calls the type discriminator on the first column.
2. The records are put into buckets by the type of the first cell, $(Null, "b")$ and $(Null, "a")$ are put into bucket 0, the **Null** bucket.
   (a) The type discriminator is called on bucket 0 on the second column, both $(Null, "b")$ and $(Null, "a")$ are put into bucket 2, the **String** bucket.
      i. The String discriminator is called on bucket 2 on column 2 character 1, $(Null, "a")$ is put into bucket 98 (in ascii an a is represented as 97, the offset of one is because end of string goes into bucket 0), $(Null, "b")$ is placed in bucket 99.
      ii. Both buckets contain only one element so they are read back into the argument bucket in order. The String discriminator returns.
   (b) The buckets are read back into the argument bucket and the Type discriminator returns.
3. The buckets are read back into the argument bucket and the Type discriminator return.
4. The setup call performs some clean up and returns **OK**.

2.3 Database benchmarking

Finally, before the system can be evaluated, a plan is needed for how it should be evaluated. In [Hoi Hei, 2016] the approach taken was to generate a database of random records of certain types, integers, numerics, 10 character strings, and 100 character strings, and then measure the amount of time it takes to execute an **ODRER BY** query at various database sizes. While this approach works as a starting point it has something of a flaw in that it doesn't mimic what an actual real world database looks like. It is rare for real world data to be perfectly distributed, and it does not mimic the types of queries that a real world database will be called on to execute. As an example, real-world databases are generally not queried for entire columns of tables but rather subsets of each column, and cross table queries where joins are used are also common. As such the data acquired can not truly be relied upon to provide accurate information on the magnitude of the improvements made or even if there are any improvements at all.

The first and perhaps most intuitive approach to producing a better measurement for database performance is to move from a simple randomly generated database and a single test query to a database generated in such a way that it mimics a real world database more closely, based on real world data. At the same time, instead of a single query, a set of queries that together are meant to represent a normal database load could be used. This was done in benchmarks such as AS^3AP [Turbyfill et al., 1993], however these usually fall short of providing representative results [Gray, 1993]. However modern, well regarded, examples like TPC-H do exist. The other possibility for

producing better measurements is to simulate a real-world system that interacts with a database. This is done in benchmarks like TPC-C.

2.3.1 TPC-C and OLTP

There are a number of different models for the load a database system should handle but the most common one is Online transaction processing (OLTP) style loads [Gray, 1993]. The typical example of such a system is a business system for a company with sells and ships items, as such the system must handle entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses, all in real time. The industry standard for simulating these kinds of systems are the TPC-C and TPC-E benchmarks, TPC-C originating in the early 90s while TPC-E is a more modern approach integrating a wider range of database uses, however the more popular one still seems to be TPC-C, potentially due to there being a number of free test runners for major database systems available for it. Both are developed by the Transaction Processing Performance Council [TPC, a] which is backed by a number of organizations that develop and produce database systems [TPC, b].

The exact details of how the TPC-C benchmark, which is the one used in this report, functions aren't truly important to understanding this report and as such they will not be discusses, just note that it is a simulation of a wholesale vendor's businesses system. However, there are a few things worth mentioning.

- The TPC-C benchmark is distributed as a specification, not as a program that can be easily integrated against a database, it falls on the test runner to implement that benchmark. This is by no means a small task and well outside the scope of what is possible to do in a project like this.

- A true TPC-C benchmark must include an audit, by a certified auditor, of the entire system used to drive the benchmark and the database system that the benchmark is run against.

- Any benchmark result that does not fulfil the above criteria can not be called a TPC-C benchmark of a system. As such when any benchmark is referred to as TPC-C in this report it is meant in terms of it being a benchmark designed to behave like a true TPC-C benchmark, without having gone through the bureaucratic process and cost of a true benchmark.

- The primary metric used for TPC-C results are new order transactions per minute. This is meant to represent the business throughput that the database system can handle and as such should be more accurate than query times for predicting usefulness to end users.

2.3.1.1 DBT2

There are a number of tools out there for running TPC-C against various database systems, such as Oracle DB or MySQL, but only one could be found that supports SQLite, and that one is DBT2[DBT, b]. As such that is what will be used in this report.

DBT2 consist of 2 primary command line utilities plus a number of supporting programs for performing post run analysis. The main components are the data generator and the test runner. The data generator is used to generate a dataset, in a csv like format, that is then loaded into the database format of choice before running. This step takes a parameter named warehouses, ranged over the natural numbers, which acts as a scaling factor, for example, if the parameter is set to 1 the dataset generated consumes roughly 80 MB of space while if set to 100 the dataset consumes around 8 GB of space. Changing the warehouse parameter is the recommended way to scale the database to measure performance at different size levels [DBT, a].

The second part of DBT2 is the test runner, this is C program, wrapped in a bash file, that runs the TPC-C benchmark, for a non-embedded database this would normally connect to a database over the net but for SQLite it has the database engine compiled into the test runner itself. Running the benchmark locally does remove the latency of over the net transactions so the results are not comparable to other database systems. This does not prevent it's use for comparing different versions of SQLite. It is important to note that due to this, whenever the SQLite configuration to be tested changes, DBT2 will also need to be recompiled against the new SQLite version or the changes will not take effect. The test runner takes a starting database generated by the data generator as an argument, runs the simulation, and at the end provides some summary statistics about the benchmark run, including the key performance metric for TPC-C, new order transactions per minute.

2.4 TPC-H

While OLTP style workloads are the most common in enterprise server based database systems, SQLite is not meant to be used as such, due to it being an embedded database system. It is still reasonable to test the discrimination based implementation on such a workload as it will give an indication of whether it might be a worthwhile change to other database systems. At the same time, the discrimination based SQLite should also be evaluated on the types of loads it will normally experience, which can be split into two types.

Firstly, loads common to medium sized embedded systems, which tend to deal with very small amounts of data, and as such will not experience much change at all from the changes that have been made, the major part of the run time there is instead taken up by constant factors such as query planning and disk access. The other major type of system SQLite is deployed in is mobile devices, as mentioned SQLite ships with every single android device. While there is no major benchmark for mobile workloads

[Kennedy et al., 2015] they often resemble business analysis workloads, sometimes referred to as decision support, in that the queries tend to be complex and focused on providing summary information of a number of tables. One major benchmark of that style is TPC-H [TPC, c], also produced by the TPC. Note that the caveats about auditing and what makes a true TPC-C benchmark also apply to TPC-H.

No TPC-H implementation for SQLite could be found so one will need to be created. Due to this the exact implementation of the benchmark becomes a lot more important than it was for TPC-C. Fortunately TPC-H is much easier to implement. For example, the TPC released, along with the specification for TPC-H, a starting dataset generator. This generator can also provide datasets for scaling factors that are not just expressed as integers but also as floating points. It should be noted that a real run of TPC-H is only allowed to be done on a limited number of scaling factors. All the information that follows is taken from the TPC-H specification [TPC, d].

Three main stages are specified, data generation, data loading, and benchmark running. As mentioned, data generation is handled by a generator that comes with the specification, and data loading is simply a matter of loading the database rows the generator created into the database. The benchmarking step consist of two different tests, the power test and the throughput test.

2.4.1 Power test

The power test consists of running two streams of queries, one after another. The streams are:

- A query stream. A query stream executes each of the 22 queries from the specification sequentially in some order. The order is given in the specification and depends on which test is being run and the number of the stream, if more than one is being run concurrently. The queries are given in a general version of SQL not designed for any specific database system and their expected run times vary greatly.
- An update stream. An update stream executes two different queries, one that pushes new rows to the database, and one that deletes old rows. The number of rows deleted and inserted are always the same. These are always interleaved and the stream always starts with the one that creates more rows. Depending on the test and the number of query streams the two queries can be run one or more times.

The power test is intended to measure the raw sequential execution power of the database system and as such the streams are run in a specific order. The steps are:

1. Execute the first query of the update stream
2. After that has finished, execute the entire query stream
3. Once that it done, execute the remaining query in the Update stream

It is scored as in Equation 2.1, where $Q(i,n)$ represents the execution time of query i in stream n, $U(i,n)$ is the same but for the update stream and SF is the scaling factor.

$$3600 * SF / \sqrt[24]{\prod_{i=1}^{22} Q(i,0) * \prod_{i=1}^{2} U(i,0)} \tag{2.1}$$

2.4.2 Throughput test

The throughput test is very similar to the power test and uses the same concept of streams but with one clear difference in that the Streams are allowed to execute in parallel. The number of query streams depends on the scaling factor, see Table2.1, these are also the only scaling factors that TPC-H officially supports. However, it is worth noting that SF 1 represents a 1.4 GB database and the scale is roughly linear.

SF	S(Streams)
1	2
10	3
30	4
100	5
300	6
1000	7
3000	8
10000	9
30000	10
100000	11

Table 2.1: TPC-H official scaling factors and corresponding number of Query streams

There is always just a single update stream but the number of updates it should run before completion is equal to the number of query streams run. The throughput test is scored according to Equation 2.2 where S is the number of Query streams and T is the total amount of time from the submission of the first query to all streams having completed. The scores are combined as $\sqrt{Power * Throughput}$

$$3600 * 22 * S * SF / T \tag{2.2}$$

Chapter 3

Roads not taken

Before discussing the experimental methodology and the results it is worth covering some approaches that were tried but did not pan out.

3.1 Discrimination sort for external sorting

As mentioned in 2.2.1 SQLite actually implements two sorting algorithms, one for external memory sorting and one for sorting elements that fit into memory. However, the discrimination based implementation of [Hoi Hei, 2016] only changes the in memory sorting algorithm to be discrimination based, while leaving the external memory sorting algorithm alone. Therefor the feasibility of implementing a discrimination based external memory sorting algorithm to replace the merge sort based algorithm that SQLite currently uses was investigated, in the hopes that the same performance increase as those seen in [Hoi Hei, 2016] might be achieved on the external sorting algorithm.

3.1.1 External sorting background

The design of external sorting algorithms is a bit different from that of in memory sorting algorithm. The reason is actually rather simple: external sorting algorithms use a different performance metric than that of in memory algorithms. To analyse algorithms a proxy for run time that can be analysed statically is needed. For in memory algorithms this is normally something like the maximum number of instructions that could be executed. However, for an external memory algorithm this is no longer truly applicable and a better proxy is the number of distinct disk accesses required, this due to the simple reason that disk access is by far the slowest part of the task. Therefore, it is possible to focus on the disk accesses as long as the processor bound part of the algorithm operates in a reasonable number of instructions, like $O(n * log(n))$. The lower bound for sorting, given the following variables, is given in Equation 3.1 assuming $B * log(m) = \omega(log(N))$, which holds for all reasonable assumptions about the machine that is being used [Vitter, 2008].

- N is the number of items to be sorted, the size varies but it is usually at least 10^{10}
- M is the number of items that can be fit into memory at any one time, a reasonable number is somewhere around 10^7
- B is the number of data elements that can be transferred at a time, this generally maxes out around 10^4 due to disk track sizes
- $n = N/B$ is the number of transfer operations required to read every item into memory once, with the above numbers, 10^6 is a reasonable starting point
- $m = M/B$ is the number of transfer orations required to fill up our memory, 10^3 is a reasonable estimate

$$2n/D * log_m(n) \tag{3.1}$$

First of all, note that this is not an asymptotic bound, but rather the absolute lowest number of disk operations required. Secondly, this number does not assume a comparison based model of sorting but rather it is based on the cost of permuting the elements, in other words, it is the minimum number of disk operations required to rearrange the elements on disk into sorted order [Vitter, 2008]. As such it also applies to distribution based sorting algorithms like Discrimination sort. This course of inquiry was therefore abandoned due to the fact that it is provably impossible to improve on the model for external sorting that SQLite uses, assuming that their implementation is correct.

3.2 Producing a drop in SQLite replacement

The second avenue explored was to produce a version of SQLite that used discrimination Sort but that had the exact same functionality, since the implementation inherited from [Hoi Hei, 2016] was known to be incomplete. This would be easy to evaluate due to SQLites comprehensive test suite, and the hope was that if such a version could be produced and shown to outperform the old version then it might be possible to get it included in the main SQLite code base. If that turned out to be difficult the plan was to instead release the new SQLite version to the public via other means, along with proof that it indeed works due to it passing the test suite.

However, after some work was done on evaluating the feasibility of doing this, and some work done on actually producing the replacement, it was discovered that there is functionality in SQLite that assumes a comparison based model for sorting. More specifically, SQLite allows user defined comparison functions for string comparison and also provides 3 different ones itself. Now there are several ways around this problem.

1. The system could be changed to instead allow user defined discriminators and changed the defaults to be discriminators, while this might have been possible, and even quite interesting to do, it is difficult to predict the scope of such a change. The SQLite code base is large, tens of thousands of lines of code, and just finding where something is defined is difficult especially for a change that

touches several parts of the API. At the same time, such a change would break the backwards compatibility with old SQLite installations, completely removing any chance that it might be included in the main SQLite code base.

2. Alternatively, it might be possible to detect when a user defined comparison function is used and switch over to the old comparison based sorting model, this is somewhat complicated by the fact that the user defined ones are passed into the sorting algorithm in the exact same way as the default comparison function, while it might be possible to detect which one is the default this depends on how SQLite has implemented this functionality. Simultaneously this complicates the code base adding complexity and overhead that for a number of users will never even add anything. As such this was judged to be a rather poor solution.

3. Finally, it might be possible to include the changes to SQLite as a compile time option, such that if a flag was enabled the user defined comparison functionality would be disabled but a faster sorting strategy could be used. This has a few benefits in that it is quite easy to do and that it does not break backwards compatibility. This option is also useful due to the fact that Discrimination sort has a different memory profile than a normal merge sort, which might cause problems for low memory embedded systems.

However, at the same time several other SQLite features that made the implementation of a Discrimination based sorting algorithm more complicated were discovered, such as the possibility of different Unicode encodings. While all of them were solvable problems it was decided that due to the complexity of the code base it was impossible to determine how long producing a replacement version of SQLite would take.

Therefore, it was decided to fix the major issues with the implementation, such as some problems with specifying a descending ordering direction and a general problem with the ordering of floating points, and then to focus on actually evaluating the implementation properly.

3.2.1 Changes made

As mentioned, the implementation of discrimination sort in SQLite from [Hoi Hei, 2016] had some problems.

First of all, there was an issue with the **DESC** tag that can be passed into an **ORDER BY** statement, support for this tag had not been implemented, and since the benchmarks that were to be performed might use it, it had to be implemented. This problem along with the others in here were discovered by first running the SQLite test suite and then through some manual debugging on a test database. The implementation of this change is complicated by the possibility to sort on multiple fields at a time, and since each one will have it's own ordering direction, either descending or ascending, which must be preserved, it is not possible to simply reverse the linked list after a column has been sorted. Instead the approach taken makes use of the following useful identity. For two lists l_1 and l_2, $reverse(l_1 + +l_2) = reverse(l_2) + +reverse(l_1)$. Using this, at the stage when the now ordered buckets are merged together, a check is made to see if

they should be reverse ordered and if so the buckets are written back into the collection bucket in reverse order.

Secondly, a problem was discovered with the ordering of numeric data types, in that an almost random ordering was returned on some systems. This was determined to be due to integers and floating points being cast to **long double**s to convert them into 80-bit floating points, see 2.2.2 for more details on this format. The **long double** datatype is specified to represent a high precision floating point, it should have at least the same precision as the **double** data type, which is a 64-bit floating point. However, what exactly it represents varies from compiler to compiler. GCC represents it as an 80-bit floating point while the Visual C++ compiler simply reads it as a **double**. A system that only works with some compilers is something that should be avoided and as such an alternative was developed that performed the conversion without casting to a new data type.

Fortunately, the code required to implement this is rather simple. For translating 64 bit floats into 80 bit floats it is possible to use a few C library functions to determine the exponent and fraction parts, which can then just be written into a byte array in the correct order. Translating an integer is a simple matter of determining its sign and exponent, the value of the exponent is simply the highest position of a 1 in the integer. The fraction is just the integer itself.

Chapter 4

TPC-C

With that covered it is time to start evaluating Discrimination based SQLite implementation. The first attempt made was to use TPC-C but after some investigation it turns out that TPC-C does not measure the changes that were made in any way.

This was discovered by inserting some code into the original version of the SQLite codebase that captured how long each call to the sort function was taking and recorded it along with the number of elements that were sorted. This method for capturing the data was chosen due to it being simple and reliable, even if it requires some small overhead for writing to a file on each call. The alternative would have been to use a profiler, which while beeing functional has some problems, more specifically, the benchmarks that were run do not just execute a binary file, instead they are executed by a bash script which uses a number of binary executable as sub programs. Therefor to run a profiler on the benchmark those bash files would need to be edited, which seems needlessly complicated.

The results from these efforts are somewhat surprising, see Figure 4.1 and 4.2. Regardless of scaling factor, the system spent around $1/10^5$ of its time on sorting and on average it sorts around 12 records at a time. This is clearly not what was expected, in fact this invalidated the results that had been collected as the changes that are being measured are so small as to be non-existent.

This result seems somewhat odd as it is reasonable to assume that a widely used database benchmark would actually exercise all the regular parts of a normal database system. Therefor the queries that the benchmark would send over the course of it's execution were examined, to identify why no change was seen in the number of records being sorted between database sizes. Out of 35 queries only one involves performing a join and no query involves using the **ORDER BY** command. That query is as follows:

```
SELECT count(*)
FROM order_line, stock, district
WHERE d_id = %d
  AND d_w_id = %d
  AND d_id = ol_d_id
  AND d_w_id = ol_w_id
```

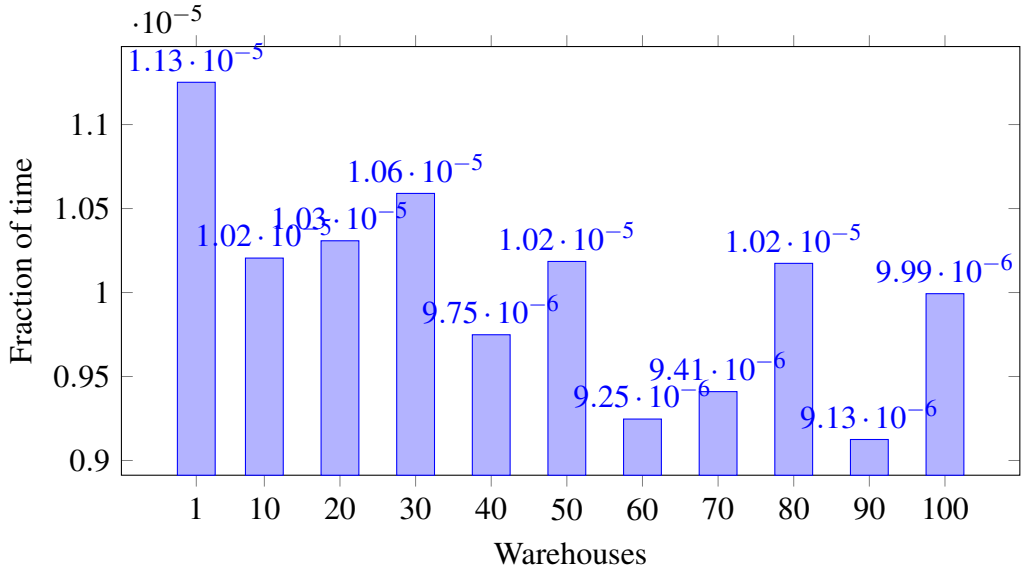

Figure 4.1: Fraction of total time spend on sorting

Figure 4.2: Average number of elements sorted per call

```
AND ol_i_id = s_i_id
AND ol_w_id = s_w_id
AND s_quantity < %d
AND ol_o_id BETWEEN (%d)
        AND (%d)
```

The %d's are part of a format string and will be substituted for different variables at run time, however what is worth noting here is that there are several filters in place that limit the number of rows that will actually be fed to the sorting engine, as SQLite will execute filtering operations before performing joins as they are much cheaper to do and can be executed together with the data being read into memory at a relatively small overhead.

While the lack of queries involving sorts is unfortunate for this project it makes sense for the type of system that TPC-C simulates, OLTP type loads are meant to be loads for real time systems, where results are required as soon as possible, such a system, if developed in the real world, would not have expensive table spanning sorts as part of it's normal workload as they would slow down the system, providing a worse user experience.

Chapter 5

TPC-H

Benchmarking using TPC-C failed to produce results that can be used to draw conclusions about the systems that were evaluated. However, TPC-H, which is more analytics focused, worked better.

5.1 Implementing TPC-H

Unlike TPC-C, no TPC-H implementation was found that supports SQLite, but as mentioned in 2.4 TPC-H is distributed alongside a data generator, and its structure is quite a lot simpler, being comprised of a number of queries to be run in a specific order. The first choice to be made is how much of TPC-H should be implemented. It would be possible to implement the entire thing, as in, both the power and throughput tests, or just one of them. It could be argued that implementing both would lend the results greater significance, but the throughput test is mostly meant to measure the system's capacity for handling multiple demanding computational loads simultaneously.

The throughput test is simultaneously a much more complex thing to tune. The obvious way to execute it, to simply have each query and update stream submit queries as fast as possible, might not be optimal due to reasons such as memory and cache loads. It also does not measure anything relevant as the changes that have made do not interact with the scheduler in any way. The number of query streams executed by the throughput test is also only defined for specific scaling factors, and they do not align with the scaling factors that are relevant to this report. As such it was decided to only implement the TPC-H Power test, this reduces both the complexity of the implementation task and also that of the results, simplifying analysis.

As for the sub parts of the power test, it consists of a query stream and an update stream, which should never be executed in parallel. The discrimination based SQLite version only differs from the original in ways that affect the query stream, as insertions into a database do not require the sorting of data unless there are indexes present, but since it is advisable to maximize the potential impact of changes made, so as to be able to measure them, no indexes will be used and they are not mandated by the

specification [TPC, d]. Therefor only the query stream was implemented, as having the update stream would not add anything to the analysis.

To execute the benchmark the following steps must then be carried out:

1. A starting data set must be generated, as per the TPC-H specification the dbgen utility that the TPC distributes is used for this purpose.
2. The data set must then be loaded into a database, apart from some minor problems the dataset is actually already in a format that can be read into SQLite, as such a bash script that creates the database and then use SQLites built in import command to import entire tables at a time is enough.
3. The SQL queries to be executed has to be converted into a format accepted by the database system to be benchmarked, as a starting point a translation of the queries to standard sql was found[Bacis, 2013]. It was then confirmed that they were consistent with the specification, and then a few SQLite specific issues were fixed, such as a few syntax errors that were found.
4. Finally, the queries have to be executed against the database, and that execution must be timed. For executing the queries a small python script was written that would load the queries into memory and then use the **sqlite3** library to execute them against a provided database. For timing there were two options, the specification could be followed and wall clock time could be used or CPU time could be used. Wall clock time is the amount of time recorded by a stopwatch while CPU time is the amount of time spent on the CPU.

   Normally CPU time is more accurate because it does not count time spend not executing, for instance the OS servicing some other process. But CPU time has the problem that it records parallel workloads doubly and since SQLite is free to use as many threads as it wants to execute queries this could complicate the analysis. Therefore wall clock time was measured as it produces a less complex analysis.

These steps were then tied together as a bash script.

While testing it was discovered that Query 20 (Potential Part Promotion Query)[TPC, d] was taking much longer to execute than any other query, in fact it was several orders of magnitude slower, and its execution time grew exponentially with the scaling factor. The amount of time required was such that it was estimated that it would take close to a day to execute it at a higher scaling factors, which would have been unacceptable. After some investigation, it was determined that this was due to two issues. Firstly, SQLites inefficient implementation of the **LIKE** query parameter. Secondly, the query performs a large scale join that if executed in the wrong order produces a large amount of rows at intermediate steps. It was also determined that it was spending a comparatively small amount of time actually sorting data, so removing it would not have an impact on the evaluation of the system. As such Query 20 was dropped from the queries to be executed to make the benchmark usable.

Since a number of parameters of the power test were changed a new scoring function was needed, for this purpose it was decided to simply use the geometric mean, as

seen in Equation 5.1, as it is recommended in [Patterson, 2011] for providing a scale invariant measure that is also easy to understand. The elements e_1 through e_m represent the n values of which the mean is being calculated. Note that a low score is better in this case.

$$\sqrt[n]{e_1 * e_2 * ... * e_{n-1} * e_n} \tag{5.1}$$

To summarize the benchmark we produced differed from TPC-H in the following ways:

- Only the power test is run, the throughput test is dropped
- Only the query stream of the power test is executed. The update stream is not.
- Query 20 was not included among the queries.
- A different scoring function was used.

5.2 Experiment Methodology

Before the TPC-H benchmark could be used a benchmarking plan was required. This task was split into two questions, which scaling factors were to be investigated and how many trials should be run at each scaling factor. Experience from TPC-C indicates that at least 10 is the answer to the latter, so 10 were used. The scores from each benchmark run were combined using the geometric mean, 5.1. Testing indicated no major outliers and as such no data was discarded.

As for the former, changes should be most visible over an exponential scale, so a wide distribution is required. Though experimentation it was found that 0.0001 is the lowest scaling factor at which it is possible to accurately measure the amount of time taken to execute all queries. At the same time, it was found that 0.5 is roughly the upper limit at which all the queries can be executed in reasonable time, which was defined as less than an hour for any query. At 0.5 it takes roughly 30 minutes to execute all queries once, but due to a number of the execution times growing non-linearly at scaling factor 1 the total time reaches 2 hours for a single pass through the queries. In the range 0.0001 to 0.5 the following scaling factors were picked, distributed on a semi exponential scale: 0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, and 0.5. This selection allows for running the full set of benchmarks in a bit less than a day.

The trials were executed as follows:

1. For each scaling factor generate 10 starting datasets using *dbgen*
2. Compile and install the original version of SQLite
3. Run the benchmark against each starting dataset
4. Compile and install the Discrimination based version of SQLite
5. Run the benchmark against each starting dataset

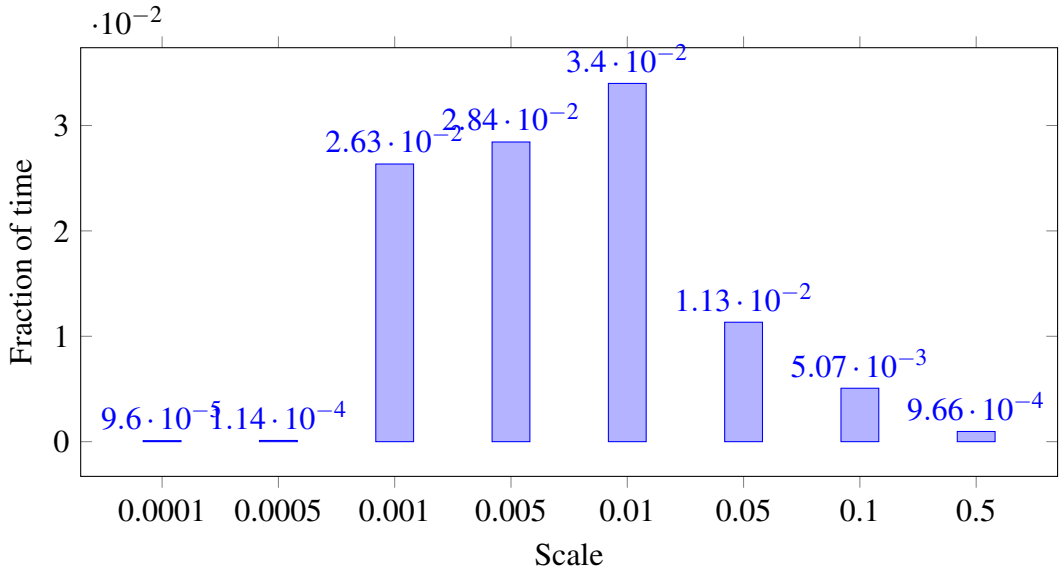

Figure 5.1: Fraction of total time spend on sorting

This ensures that the benchmark is run against the same database's for both versions of SQLite while minimizing overhead due to the cost of recompiling SQLite.

5.3 Analysis

At first it was confirmed that it might actually be possible to detect changes by checking how much time is spent on sorting for each scaling factor using the same method as used in Chapter 4, see Figure 5.1 for the results. The fraction of time spent on sorting is clearly larger than before, and for a few scaling factors it might be possible to detect changes too it, but it still seems low. However, a few of the queries take a disproportionate amount of time to execute, and this data is normalized over the full amount of time it takes to execute the full set of queries. If some queries spend a large fraction of their time sorting but overall have short execution times then those would be underrepresented here, but it might be possible to still detect the changes when the score is calculated. If the time spent sorting is instead broken down by both scaling factor and by query number, see Table 5.2, this does indeed seem to be the case, several queries spend a large portion of their execution time on sorting, which means that it should be possible to detect changes in the sorting execution time.

However as seen in Table 5.1, at scaling factors 0.0005 and below the system only sorts a couple of elements on average. It is not possible for such a small number of elements sorted to have an overall impact on the results, which is confirmed by the results in Table 5.2, as the highest fraction of time spent on sorting at a scaling factor of 0.0005 is 0.018. As such, results obtained at those scaling factors can not be considered reliable and were discarded.

Since there is cause to believe that changes could be detected the results were examined, see Figure 5.2. Merge sort was found to outperform Discrimination sort at every

Scale	Elements
0.0001	2
0.0005	2
0.001	471
0.005	1924
0.01	3538
0.05	13786
0.1	20193
0.5	31795

Table 5.1: Scaling factor and average number of elements sorted

Scale	1	4	9	10	11	15	16
0.0005	0.018	0.000	0.000	0.000	0.000	0.000	0.000
0.001	0.377	0.015	0.045	0.064	0.000	0.000	0.041
0.005	0.366	0.012	0.048	0.070	0.010	0.001	0.063
0.01	0.339	0.014	0.053	0.063	0.013	0.004	0.075
0.05	0.355	0.015	0.059	0.060	0.012	0.013	0.098
0.1	0.350	0.016	0.029	0.060	0.010	0.019	0.096
0.5	0.363	0.019	0.016	0.058	0.003	0.030	0.120

Table 5.2: Fraction of total time spent sorting. Columns with no values above 0.01 removed for readability

scaling factor. The scale is also flat with no discernible changes as the database size goes up. This was not what was expected as the asymptotic run times indicated that they should grow differently with the number of elements. There are a couple of different possible reasons that this could be the case. First of all, it might just be that the different in growth is so small that nothing changes over the range that is examined here. Or it could be that the number of elements being sorted does not grow as the Scaling factor grows.

The former is unlikely, the difference in database size is a factor of 500, and if no change is apparent after that kind of change then any change in ratio is functionally nonexistent, even if it is there theoretically, it is also difficult to investigate since running the benchmark at much higher scaling factors would result in total run times that are too long.

The latter is a lot easier to investigate as it is just a matter of recording each call to the sort function, see Table 5.3. The total and average number of elements sorted per run does increase as the Scaling Factor increase, however the average number does not increase as much as might be expected, it is certainly less than a linear increase. The reason appears to be that the number of calls to the sort function goes up instead, with the maximum number of elements sorted remaining stable for the later scaling factors. This can only mean that the external sorting algorithm is being invoked, as such SQLite's default configuration considers around 37000 elements to be the maximum that it can hold in memory at a time.

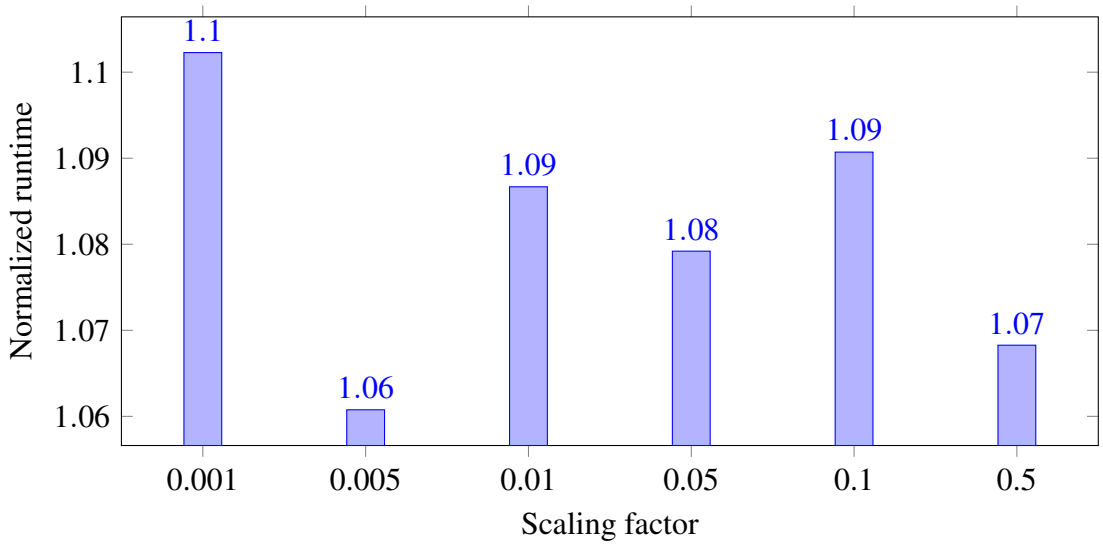

Figure 5.2: Normalized runtimes. Results below 1 would indicate Discrimination sort performing better

Scaling Factor	Average Sorted	Max Sorted	Number of sort calls
0.001	471	6005	15
0.005	1924	30201	19
0.01	3538	37908	22
0.05	13786	37929	28
0.1	20193	37940	38
0.5	31795	37942	118

Table 5.3: TPC-H sort scaling. Average and max is per call

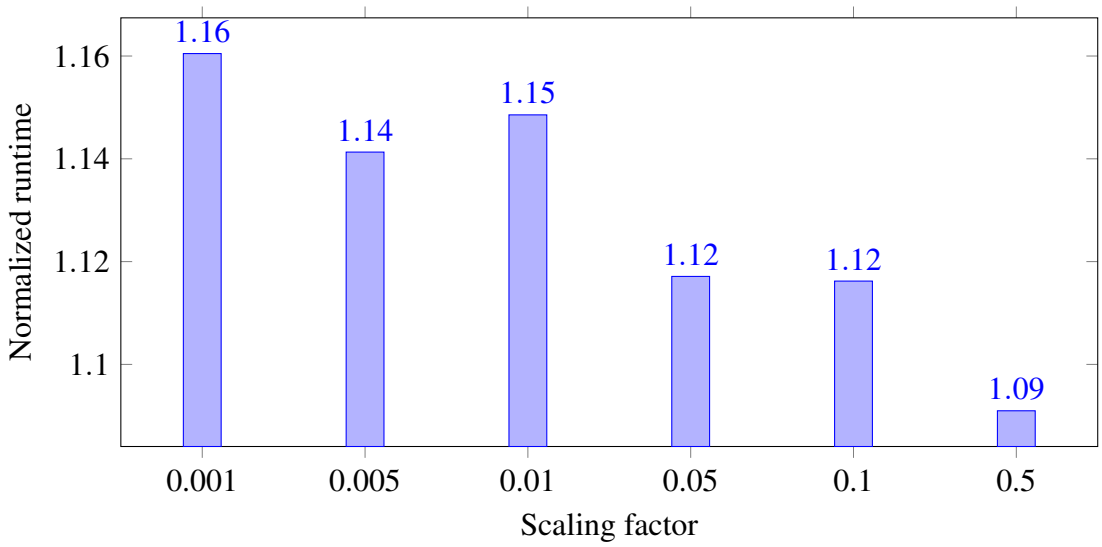

Figure 5.3: Normalized runtimes. Results below 1 would indicate Discrimination sort performing better

A maximum of 37000 is relatively low all things considered, at least for a desktop or server machine, but SQLite is also meant to be able to run on embedded systems, and in that case, it is a significant amount. This does however mean that if the point where discrimination sort starts outperforming merge sort is outside of that range, then it will never be detected unless the default is changed. It is possible to change the default as it is calculated based on the cache size, which is a parameter that can be changed at run time, the default value is roughly 20 MB. Doing so would however to some extent invalidate any results obtained. The goal was to investigate if using discrimination sort would improve performance in databases, and the conclusion here is the opposite, the performance is in fact reduced for a large number of cases. Even if a performance increase is found if the parameters are changed that would still mean that every single user who does not change the cache size gets a significant performance reduction.

However, not all database systems are intended for the same purposes as SQLite and they therefor use higher defaults for these kind of limits and results obtained on SQLite should provide an indication of how the performance might scale. The same tests as outlined above were therefor carried out with a cache size of 4 GB, 4 GB being the upper limit of what was determined to no fill out the memory of the testing machine completely.

See Figure 5.3 for the results. The ratios are a bit higher than before due to the reduced overhead on reading elements from disk, so they can not be compared directly to results from where the cache size was lower. The distribution is also very different from earlier, instead of a flat scale there is a clear downwards progression, indicating that Discrimination sort does indeed scale better with the length of the input than merge sort. However, it is still at best 9% slower than merge sort. It is possible that there exists a point where discrimination sort would catch up to merge sort, but it is not found within a range that could be tested using available resources.

Chapter 6

Manual investigation

The results in 5.3 contradict previous knowledge [Hoi Hei, 2016] and that is worth investigating. At first it was noted that the results presented in Hoi Hei [2016] were only in the range of 1 to 46 million records. It is very much possible that the point where Discrimination sort becomes preferable lies somewhere below this range. Examining the code used for those benchmarks it was found that they were run entirely on an in-memory database, with no disk interaction. As such there is no cap to the number of elements that are fed to the internal sorting algorithm in this setup. It was therefor decided to repeat the same benchmarks but on a different scale.

For the upper limit the largest database that could be fit into main memory was used, which after some testing was determined to be 70 million rows on a machine with 16 GB of ram available. Due to the low cost, it was possible to take measurements of the entire range at an exponential scale. As such the following benchmarks were run.

For each $m = 1.5^n < 70,000,000$ where n is an integer, measure the amount of time it takes to sort m random strings (length 1 to 10), 64-bit integers, floats $\in (-1,1)$, and numerics defined as a mix of 64-bit integers or a floats $\in (-1,1)$. Repeat 5. 1.5 was used as the base of the exponent due to it providing a good balance between measuring resolution and total runtime of the tests, determined through testing. The results are displayed in Figure 6.1, 6.2, 6.3, and 6.4 respectively. Points below 1 indicate discrimination sort performing better. The data from the separate runs were again combined using the geometric mean, see Equation 5.1.

The low end was discarded for each data set as the results are unstable due to the small time steps making the measurements unreliable, however outside this range the variance is low and as such no discarding of outliers was necessary. The difference in performance between the different data types relative to the Merge sort implementation can be explained by constant differences in time required to either compute the next byte to use for Discrimination or the cost of comparison. In order:

1. In the case of strings, as mentioned in Chapter 3.2, the discrimination based implementation has several simplifications that reduces its overhead compared to the merge sort based implementation.

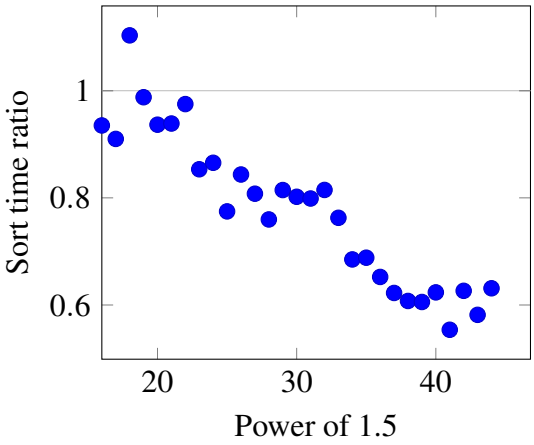

Figure 6.1: Sorting strings

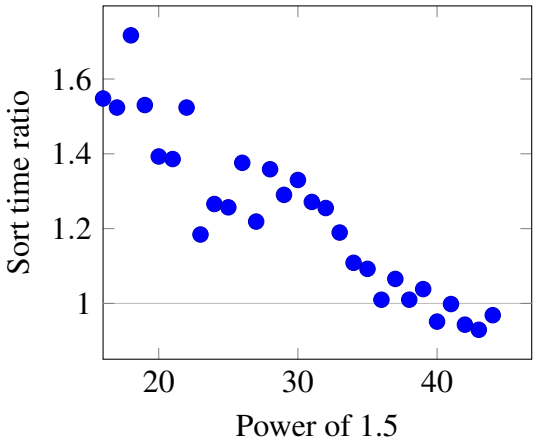

Figure 6.2: Sorting Integers

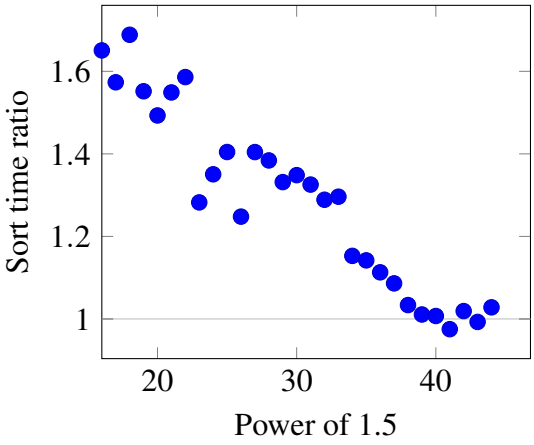

Figure 6.3: Sorting Real Numbers

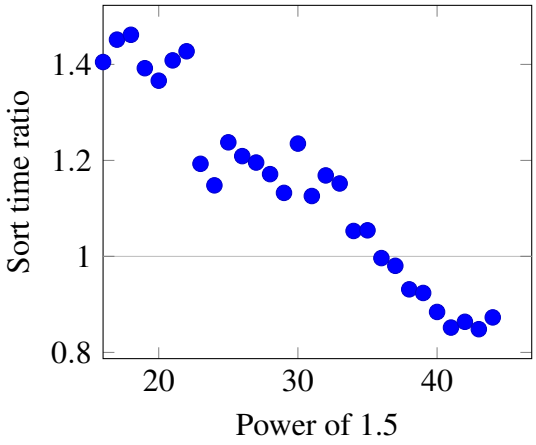

Figure 6.4: Sorting Numerics

2. For integers and reals comparison is highly efficient and the merge sort implementation does not even have to decode the elements to compare them [SQLite, e], while the discrimination based implementation maps them onto 80 bit integers which is a comparatively costly process.
3. For a mix of integers and reals the merge sort implementation has to sometimes decode the elements, so the average cost of comparison is increased. At the same time the constant factors for the discrimination based implementation stays the same, so the gap in performance is reduced compared to reals and integers.

In conclusion, the discrimination based implementation does outperform the merge sort based implementation at high loads but at low loads there is a definite performance loss. The cross over point is dependent on the exact datatype used. As such, discrimination sort is a promising development for database systems, but care must be taken to not apply it to systems where it has a high potential to reduce performance.

Chapter 7

Conclusions

This project explored discrimination based sorting in database systems as an alternative to comparison based sorting. SQLite, a widely deployed embedded database system, was used as a reference database system. Most of the report is spent on evaluating discriminators potential to be used for in memory sorting since, as seen in Chapter 3, it is theoretically impossible to improve upon the asymptotic run time of commonly used external sorting algorithms.

It was found that for common business oriented workloads (OLTP workloads) the sorting algorithm makes little difference, see Chapter 4, as the number of elements sorted does not scale with the size of the database due to the simplicity of the queries such a workload entails.

The sorting algorithm used does however have more bearing on other types of workloads, especially those that are more analysis focused, sometimes known as decision support workloads. These are most commonly used in big data scenarios and in mobile devices. In Chapter 5 the merge sort implementation was found to outperform the discrimination sort implementation on decision support workloads at all database sizes that were tested. However, once the upper limit for the number of elements that could be sorted at a time had been increased there was some indication that discrimination sort might outperform merge sort for large enough databases with access to enough RAM. This could not be tested due to a lack of available computing resources.

These results did however contradict previous work [Hoi Hei, 2016] that found a consistent performance increase with discrimination sort over merge sort, but on further investigation it was found that those results were due to an insufficiently small measuring window, see Chapter 6. When the same experiments were repeated with a different measuring interval it was found that while discrimination sort does indeed outperform merge sort at sufficiently large database sizes it does not do so at every size.

In conclusion, Discrimination based sorting does show promise in being applied to databases and can result in overall performance increases depending on the size of the database and the amount of resources available to it. However, discrimination sort can not be recommended for SQLite specifically since SQLite is not intended to be used for databases of a size where the improvement start to become relevant. Any database

system that seeks to use discrimination sort should evaluate the choice based on the most common use cases for said database system as it is otherwise possible that a large subset of users would experience a performance decrease.

Future work on the subject could explore any number of different of paths. One option would be to apply and test discrimination sort on a database system that is meant to handle more data than an embedded system like SQLite. Another path that might be worth exploring is to produce some hybrid approach between discrimination and merge based sorting, since there is a threshold where discrimination sort starts outperforming comparison based algorithms. This could potentially produce a system that has the benefits of discrimination sort when there is a lot of data but that does not slow down on smaller amounts. A similar approach is sometimes used for large scale external sorting Vitter [2008]

Bibliography

Enrico Bacis. Tpch queries. https://github.com/enricobacis/jupiter-py/tree/master/tpch/queries, 2013. Downloaded: 2017.

DBT. The dbt2 source. http://osdldbt.sourceforge.net/, a. Downloaded: December 2016.

DBT. Database test suite. http://osdldbt.sourceforge.net/, b. Last accessed: 19-3-2017.

Jim Gray. Database and transaction processing performance handbook., 1993.

Fritz Henglein. Generic top-down discrimination for sorting and partitioning in linear time. *Journal of Functional Programming*, 22(03):300–374, 2012.

Ma Hoi Hei. Applying generic discrimination to achieve asymptotic improvement in databases, 2016. Honours Project at University of Edinburgh.

Oliver Kennedy, Jerry Ajay, Geoffrey Challen, and Lukasz Ziarek. Pocket data: The need for tpc-mobile. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 8–25. Springer, 2015.

David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

SQLite. About sqlite. https://www.sqlite.org/about.html, a. Last accessed: 6-3-2017.

SQLite. Data storage. https://developer.android.com/guide/topics/data/data-storage.html, b. Last accessed: 6-3-2017.

SQLite. Datatypes in sqlite version 3. https://www.sqlite.org/datatype3.html, c. Last accessed: 6-3-2017.

SQLite. Database file format. https://www.sqlite.org/fileformat.html, d. Last accessed: 6-3-2017.

SQLite. Sqlite source (version 3). fossil clone https://www.sqlite.org/src sqlite.fossil, e. Downloaded: 2016.

SQLite. Most widely deployed and used database engine. https://www.sqlite.org/mostdeployed.html, f. Last accessed: 6-3-2017.

TPC. Tpc: Benchmarks. http://www.tpc.org/information/benchmarks.asp, a. Last accessed: 14-3-2017.

TPC. Tpc: Who we are. http://www.tpc.org/information/who/whoweare.asp, b. Last accessed: 14-3-2017.

TPC. Tpc-h. http://www.tpc.org/tpch/, c. Last accessed: 29-3-2017.

TPC. Tpc benchmark™h (decision support) standard specification. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.1.pdf, 2014d. Revision 2.17.1.

Carolyn Turbyfill, Cyril U Orji, and Dina Bitton. As3ap: An ansi sql standard scaleable and portable benchmark for relational database systems., 1993.

Jeffrey Scott Vitter. Algorithms and data structures for external memory. *Foundations and Trends® in Theoretical Computer Science*, 2(4):305–474, 2008.