Mälardalen University

School of Innovation, Design and Engineering

Master Thesis in Computer Science

Optimizing the execution time of SQLite on an ABB robot controller

Author: Björn Delight bjorn.v.delight@gmail.com

Mälardalen University Supervisor: Dag Nyström

ABB Robotics Supervisor: Patrik Fager

Examiner: Mikael Sjödin
1. Abstract

The purpose of this thesis is to generate a set of general guidelines to improve performance of the SQLite database management system for use as a component in the main computer of ABB industrial robots. This was accomplished by measuring the factors that affect query execution time and comparing SQLite features to systems it could potentially replace: chiefly ABB’s configuration storage system and real-time logging system. Results show that SQLite is slower than ABBs current configuration system and real-time logger. A series of tests were carried out that give a rough description of SQLite’s performance. Read and update become significantly faster when used with an index, write becomes a bit slower. In an indexed database the number of rows is not important; in a database without an index they cause significant slowing. Surprisingly, increasing the number of threads concurrently accessing a database had only a mild effect on timing.
# Table of Contents

1. Abstract ................................................................................................................................. 2

2. Introduction ............................................................................................................................ 6
   2.1 Research problem ............................................................................................................... 6
   2.2 Contribution ....................................................................................................................... 7

3. Background ............................................................................................................................. 8
   3.1 Basic theory ....................................................................................................................... 8
      3.1.1 Database Management System .................................................................................. 8
      3.1.2 Real-Time ................................................................................................................... 8
      3.1.3 Multithreading .......................................................................................................... 9
      3.1.4 Concurrency and durability ..................................................................................... 9
   3.2 SQLite .................................................................................................................................. 9
      3.2.1 Overview .................................................................................................................. 10
      3.2.2 Locking scheme ....................................................................................................... 10
      3.2.3 Journal ..................................................................................................................... 10
      3.2.4 SQLite data structure ............................................................................................... 11
      3.2.5 Searching ................................................................................................................. 13
   3.3 Comparison between the features of CFG and SQLite ....................................................... 13
      3.3.1 Searching ................................................................................................................ 14
      3.3.2 Multithreading ......................................................................................................... 14
      3.3.3 Structure ............................................................................................................... 14
      3.3.4 Performance ............................................................................................................ 14
      3.3.5 Encryption and Locking ......................................................................................... 14
      3.3.6 Conclusion .............................................................................................................. 14

4. Performance .......................................................................................................................... 15
   4.1 Execution Time ................................................................................................................. 15
   4.2 Optimizing SQLite ............................................................................................................ 16
   4.3 SQLite with Soft Real-Time ............................................................................................ 16
   4.4 What effects the execution time of SQLite ....................................................................... 16

5. Method .................................................................................................................................... 18
   5.1 Model ............................................................................................................................... 18
   5.2 Experiment design – Use Cases .................................................................................... 18


2. Introduction

ABB robotics is a business unit of ABB that designs and manufactures industrial robot arms. An ABB robot consists not only of a mechanical arm with motors but also a cabinet that contains power electronics and the computers that control the robot arm; one of these computers, the head computer, controls networking, storage of control sequences, and an interface with the user among other things. The robot system can be considered to have two modes. A “real-time mode” where the robot is moving or even just running a user program, and a “non-real-time mode” when starting up or writing configurations; in real-time mode there are a series of hard deadlines which may not be missed, for example if the system detects that a user is too close to the robot. The robot has a strict deadline by which it must stop. If a deadline was missed it would be a failure of the system, might have legal consequences and could jeopardize human safety.

Not all tasks require strict deadlines. Among those that do not are the tasks that run during system start-up or configuration, in general anything that happens before the robot arm has started moving runs in non-real-time mode. While missing a deadline might not cause someone to lose a limb the customer should still feel that the system is “responsive”, in other words that system performance is still important. One of the main tasks in non-real-time mode is saving configuration data. Configuration data includes MAC addresses, network names, IP addresses and calibration constants. All these are saved in a file/data format called CFG. CFG, in addition to being a file format is a series of functions for simple linear searches and fast access of known entries. CFG, while fast, is not much better than an array since it lacks indexes and foreign keys (links between entries in the database). Another drawback is that it is effectively impossible to find a developer with experience in CFG outside of ABB robotics. The entire CFG system could be replaced by a SQLite based system. The problem is execution time: SQLite can be a very slow, but with the right configuration and database shape it may be possible to make SQLite feel responsive enough. Uprobe on the other hand, being a real time logging tool, has to write in real time, but only has to read to save logs to disk. However both write and read time are important for CFG. Finally since one task may be writing to a configuration database while another task reads from it; the execution time for multiple threads accessing a database is also important.

Among the programs that run in real-time mode is a special system for recording debugging messages called Uprobe. Uprobe records debug statements with timestamps and other information during real-time operation and only saves in non-real-time. Ideally such a system would have no effect on the timing of the system, so it is designed to be as fast as possible. The output of Uprobe is just a text file of debug statements. An SQLite based system would allow for far more informative access, allowing the user to look for entries which came after a specific time, the entry with the largest numerical value or simply all entries with a given context or message.

2.1 Research problem

ABB currently uses CFG as a data storage system. CFG is just a linked list without built in concurrency control, any sort of searching aids or other database functionality. Another system that could potentially be replaced is the logger Uprobe, which runs in real-time, meaning if it misses a deadline the system has failed. Both of these in house systems could be updated with ready-made open source solutions that
deliver far more functionality and portability. ABB has already started moving less time critical parts of the software to SQLite (without prior testing). However can an SQL database really compare to the speed of the systems in use? In order to make SQLite a viable replacement it must be made as fast as possible. Therefore this thesis sets out to create a set of guidelines for the use of SQLite in an ABB robot controller in a way that optimizes performance. This was accomplished by creating and then running a series of benchmarks that reflect the planned use of SQLite.

2.2 Contribution
The primary contribution of this thesis is a general description of the performance of SQLite when it runs on the ABB robotics main computer. To optimize real-time logging in addition to the saving and loading of configuration data a wide array of benchmarks where performed with different parameters. The following areas were investigated:

- SQLite performance, how do the following effect execution time:
  - The shape of the data stored: number of rows, number of columns.
  - Various settings: write ahead logs, indexes and so forth.
  - Number of threads reading and writing to the database.
- A comparison of the features of CFG and SQLite
- An evaluation SQLite’s of real-time performance via a test with Uprobe. The old circular buffer backend and a new SQLite backend are compared.
3. Background
This thesis is primarily concerned with the computational performance of SQLite. The basic theory section explains a few basic ideas that are referred to in the rest of the Background and related work as well as the rest of the thesis. SQLite gives an overview of what SQLite is as well as a few details on its inner workings. The performance section discusses methods for measuring execution time as well as look at previous work directly relevant to this thesis. Finally the use cases section outlines the process of turning use cases into concrete tests.

3.1 Basic theory

3.1.1 Database Management System
A database management system is a useful abstraction layer between application and data, the general architecture can be seen in figure 1. A database management system allows user programs to access data without having to think about its organization and structure as well as providing utilities such as a fast search. One of the most common data models for databases is the relational database model, which organizes data in several tables; tables in turn consist of columns and rows [1]. The database management system acts as an abstraction layer in front of the database, making it more or less a black box that returns data in response to queries. A query is not necessarily a question but an access to the database, either reading an entry, adding an entry, updating an entry, deleting and entry or altering the database structure in some way (creating tables or adding columns for example). In a perfect world the external schema would have nothing to do with organization of the data inside the database. However like all abstractions the one provided by the database management system forces the user to confront elements of the implementation. This means that the behavior of a database management system can be better understood by looking at how it allocates memory, maintains and modifies data structures as well as how it make access plans based on queries.

3.1.2 Real-Time
Some of the research problems attempt to use SQLite in a real time system. But what is a real-time system? A real time system has hard deadlines that must be met. Missing a real time deadline is generally considered a system failure (except in soft real time where it merely causes degradation in quality of service, for example telephones where a missed deadline merely means a lag in audio) [2]. The computer programs that run on PCs and mobile phones, that is to say those most people directly interact with, do not usually have hard deadlines, instead these devices merely attempts to deliver a result as fast as possible. When it comes to real-time systems and other systems that interact with the real world, especially safety critical ones, this “best effort” approach is often insufficient. There must be a guarantee that the task will be finished before a given deadline. While a real-time system often must be very fast, the most important thing with a real-time system is that it is predictable, it must make the
deadline every time [2]. This means that one of the most important factors in real time systems is jitter defined as difference between best and worst case execution time.

### 3.1.3 Multithreading

Ideally a real-time system would run a single process on a single processor, unfortunately for more complex systems the inefficiency of doing so becomes impossible to ignore. So real time systems often employ a real time operating system. A real time operating system is an operating system that is designed to meet the requirements of a real time system. Just like non real-time operating systems the real time operating system divides processor time between several tasks running on the same machine. On a normal computer each program is given a slice of execution time based on priority. The real time operating system VxWorks as it runs on ABB robots only allows the highest priority task to run; lower priority tasks have to wait for the higher priority ones to finish; however when two or more tasks are of the same priority, VxWorks uses a round robin scheduler where all processes of the same priority get an equal slice of time [3].

### 3.1.4 Concurrency and durability

A major issue in multithreading is concurrency. Concurrent mean simultaneous, in other words the behavior of databases when the same data is accessed from several threads or processes, a database must be able to handle concurrency safely. Multithreading introduces a series of problems related to the sharing of resources. Take file access for example: if two tasks both read a number from a file, add one to it and write it back to the file. Then the value the variable has depends on whether the first task managed to write back before the other could read it. This is called a race condition, the name comes from the fact that the outcome depends on which process wins the “race” to access the resource. In order to prevent this and other concurrency complications most operating systems (both real-time and not) assure that those snippets of code that are susceptible to race conditions cannot run at the same time. This is done with mutual exclusion flags, or MUTEXs which guarantee that only one task is accessing a resource at any one time [2]. A related consideration is how the database handles unfinished queries, in other words what happens if a power outage occurs or the task is otherwise unable to complete its query. The methods SQLite uses for dealing with concurrency and durability will be covered under the locking scheme and journal sections.

### 3.2 SQLite

Certain details on how SQLite works are interesting for this thesis and are explained in this section. SQLite is not designed as a real-time database management system; merely a small and self-contained one. [4] used SQLite in as a soft real-time database by grouping inserts and by using an in-memory database (that is to say a database that resides in RAM not on the hard drive), in memory databases in older versions of SQLite exist only for a single process, but [4] created a file system in RAM via the operating system, VxWorks and Linux have similar functionality. Newer versions of SQLite also support multithreaded (but not multi-process) access to databases.
3.2.1 Overview
SQL stands for Structured Query Language, it was developed at IBM research labs by D. Chamberlin and E. F. Codd. It implements a language to interact with relational databases and was standardized by ISO in 1987, it is something of an industry standard [5]. SQLite, the database management system that is the focus of this thesis is a small database that can be embedded in an application and requires little maintenance once it is up and running. SQLite has a very good multiplatform support. SQLite is tested on Windows, OS/X and Linux, and is ported to VxWorks. A separate module for the operating system interface, called VFS [6], facilitates easy porting. In this section and the next, basic information about its locking scheme and features is presented. While many of the important details of implementation are covered in the performance section, the locking scheme is covered here in some detail. SQLite is an open source database engine that uses the SQL standard as an interface. SQLite’s self-contained nature makes it a popular choice for embedded systems [7]. The SQLite organization tests these things using a series of test benches and claims to test 100% branch coverage of the SQLite code [8]. SQLite has no explicit support for VxWorks (and the authors had to make a few minor changes to the source code to get SQLite to run the tests) It should be noted that only elementary correctness control was performed in this thesis.

3.2.2 Locking scheme
If several threads attempt to access the same database at the same time without any oversight they can easily cause problems for each other, see the section on concurrency and durability. SQLite makes sure several processes do not access the same database in an unsafe manner with a series of MUTEX locks. This is accomplished by preventing a task from making changes to the database while another task is reading or writing. The standard SQLite locking system involves five locking states: it can be UNLOCKED which means no one is trying to access the database; it can be SHARED, which means that one or more tasks is reading the database so no writing may occur; it can be PENDING in which case a task wants to write to a database file, but there are still tasks reading it, when in this state no new SHARED locks can be open; furthermore it can be RESERVED, which means that a task will write to the database but wants to make sure no other tasks take the PENDING lock; lastly it can have an EXCLUSIVE lock which means no other tasks may read or write the file, a task must acquire an EXCLUSIVE lock in order to write to the database [9].

3.2.3 Journal
SQLite assures data integrity with a rollback journal, when SQLite starts to make changes to a database is first makes a journal of those changes if the task is unable to complete its transaction the journal is “hot”. Before a task can acquire a read or write lock SQLite checks the rollback journal, if it is still “hot”, a rollback is performed returning the database to its previous state [9].

One problem with this locking system is that when a task writes to the database no other tasks can read the database. One solution, used by [4] is breaking the database into several files or blocks. First the writing task writes to a block and when it is done it hands the block to a reading processes that have a group of blocks it can read, when they are done, the database is deleted. This locking scheme has a tendency to lock more than necessary, with accompanying losses to performance, but there is an alternative.
Another option for assuring durability is a write ahead log (WAL). The reading process writes to a WAL instead of the database file, allowing reading tasks to write without interfering with the reading tasks, the downside is that every time a task has to read it has to check the WAL to make sure it is not modified there. This means that the locking behaves a bit differently from the way mentioned above. The WAL has to be written to the database eventually, but this can be done while other tasks are reading. The writing task just stops writing at any point it comes across another task reading the same database and waits for the task to finish reading. When the WAL is completely written to file the WAL is truncated [10]. Disadvantages to WAL include slightly slower read time (readers must check the WAL in addition to the database), large transactions are slower and an extremely large transaction might not work at all. However using a write ahead log it is possible to write and read at the same time without the overhead of attaching and detaching databases.

B-tree (d=2)

3.2.4 SQLites data structure

SQLite organizes its data in a data structure called a B-Tree [6]. The B-Tree was introduced by Bayer and McCreight while working at Boeing Scientific Research Labs and is something of a standard for database structures and even file systems [11].

A B-Tree is a tree similar to a binary tree; however instead of each node having one data point and two children each B-Tree has d data points (where d is the degree of the B-Tree) and d+1 children, see figure 2 for a not quite full B-Tree. In a B-Tree each data point is a primary key, that is to say a column that uniquely identifies an entry. Since the key is unique the database entries can be ordered by its key, searching through the list would be time consuming, and there is a better way. Each node in the B-Tree can have up to d keys separated by d+1 pointers to child nodes, all members of the child nodes are between the values of the nodes that neighbor it. The leaf nodes have no pointers, only keys [11].

This means that the key is integral to the structure and cannot be changed haphazardly, changing the key is essentially deleting an entry and then creating a new one, and can trigger a rebalance of the tree [11]. The advantage of a B-Tree is that searching for a specified entry is closer to O(\log(n)) where n is the number of entries in the database. However since a node in the B-Tree is a short array the execution time is somewhere between O(\log(n)) and O(n). However if all the entries are to be accessed, every node has to be visited anyway, so the complex structure of a B-Tree means it takes more time than if the data had been stored in an array [11].

If a key is to be inserted then the tree is searched until an appropriate place in the tree is found, if the leaf node where the key would be inserted is already full the node is split into two nodes, since this requires a new entry in the parent node the split can propagate all the way to the root node. When this occurs the old root node must be split in two, and a new root node created, making the tree one level deeper, so called “Tree splitting”, illustrated in figure 3. B-tree splitting in this fashion can take quite a while, and makes any query that happens to force a split a lot slower. Furthermore, to get true
logarithmic performance, the tree must occasionally be balanced, which is also time consuming and may involve memory allocations. Due to tree splitting and rebalancing, the time taken to insert or delete entries in the database cannot be predicted without knowledge of the structure of the B-Tree at that time.

SQLite uses a standard B-Tree, divvied into pages of memory. The B-Tree is balanced after every query (though the balancing algorithm may do nothing more than double check the B-Tree is balanced). The back end of SQLite is essentially a standard B-Tree [12].

**Tree split**

![Tree split diagram]

*Figure 3: B-tree split.*
### 3.2.5 Searching

One of the most important features of a database is its ability to search through rows for the value specified by a query. As discussed above SQLite uses a B-Tree which allows it to search with close to logarithmic search time. Of course this only works with the primary key, that is to say the value the B-Tree is organized by. Fortunately SQLite can create an index on a specified column of a table, SQLite then creates a new B-Tree populated not by data but by references to the indexed table. [13]

![Figure 4: A tree index.](image)

Figure 4 gives an example where the data tree is organized by age but has an index tree for names. This allows searching of the data by name by searching the name index and then following the link to the actual data tree allowing a fast search of, for example, the age of Anthony. The downside of indexes is that having two trees means that writing takes more time, as does updating anything that would change the index. Therefore the presence or absence indexes and how they changes execution time is of great interest when measuring execution time in SQLite.

### 3.3 Comparison between the features of CFG and SQLite

The system most likely to be replaced by SQLite is the one that manages configuration files used to store information about the system. The so called CFG system is a legacy system based on a series of linked lists that reside in RAM, though they are moved onto disk automatically when power is lost. Since CFG is little more than a wrapper for a linked list it outperforms SQLite when it comes to execution time. However since the configuration system is not used in real-time, there is no danger of a SQLite system that replaces it missing a deadline. An overview of the other differences between SQLite and CFG, demonstrates larger number of features when it comes to SQLite. The information here is based not only on the source code and documentation for CFG, but also conversations with my ABB advisor, Patrik Fager.
3.3.1 Searching
In order to access an entry in CFG either the entry ID must be known (though once the ID is known access is via a hash table) or the programmer has to write code for searching through all instances which means retrieval time is linearly proportional to the size of the database [14]. While SQLite has its own functions for searching any column and support indexes for fast searching on frequently searched columns giving search times logarithmically proportional to database size.

3.3.2 Multithreading
CFG exists in the persistent memory pool and is accessible from any thread, and protection of those entries which are not write-locked is left to the programmer [14]. While SQLite has support for multithreaded and multiple process access (though multiple processes may suffer significant slowdowns as compared to multiple threads since this force the use of file systems in memory which are slower than normal in memory databases).

3.3.3 Structure
CFGs structure consists of domains at the highest level, each of which is exported as its own file and consists of several types, which are equivalent to tables; containing the structure of the instances they contain [14]. SQLite has the standard SQL structure of databases tables (which possess columns) and rows; in addition SQLite is a true relational database and has full support for foreign keys as well as functions such as JOIN, COUNT, and MAX among others.

3.3.4 Performance
Since CFG directly accesses memory through a hash table [14] it is almost certainly faster than SQLite with its many abstraction levels and complex data structure. Probably on the order of one tenth the execution time if Uprobe is anything to go by.

3.3.5 Encryption and Locking
Encryption is useful for protecting company confidential information while locking makes sure certain values are not changed. CFG has internal files which are only accessible from the ABB programs. Besides write protection [14], encryption is handled during the loading and saving of CFG files, confidential files are set as internal, and not accessible by users. While SQL files can be set as write only; most locking and encryption must be done by the programmer.

3.3.6 Conclusion
When it comes to multithreading, structure and searching SQLite was superior; CFG and SQLite where identical in encryption and locking; while CFG had the advantage in performance. When comparing CFG and SQLite it was clear that SQLite has more features than CFG. On the other hand CFG lacks the extensive overhead of SQLite, and is in general a leaner system. However since CFG is not used in true real-time, SQLite still seems viable.
3.4 Performance

3.4.1 Execution Time

When it comes to real-time systems, the only execution time of importance is worst case execution time, or WCET, which is the maximum (or worst case) amount of time a task takes to calculate its result, at a higher level worst case response time is the number to consider, but it depends on the interplay between the threads and WCET. WCET is the easiest to study and the best indicator of SQLite performance; of course any field implementation with real time requirements would first require worst case response time tests.

The most rigorous evaluation of WCET is via a syntactic tree, and while such a tree could be created for the entire SQLite program (all 140.495 lines of code), the actual data access patterns and how this is reflected in execution time is difficult to model. This method was not attempted in this thesis. Nevertheless it may be doable for specific, strictly defined cases.

Real-time performance is important not only so that important deadlines are not missed but also because execution cycles are a limited resource and poor measurements can lead to allocating to many resources to a task which does not require them [15]. Moreover if performance is measured to be worse than it actually is, a sub optimal solution may appear necessary.

How can execution time be measured? There are a number of methods, which should be used, depends on the importance of measurement resolution and the resources available:

- A common approach is to register the time when you start a task and when it ends, this of course involves an overhead in calling the system and setting the clock, one way to compensate for this is to run so many tests in a row that the error due to timing overhead is insignificant. On the other hand optimizing execution time requires only relative performance. Therefore this, “stop-watch” method is used for most of the tests in this thesis.
- The most scientific method is probably static analyses, which involves finding the “longest” path through the code and counting cycles per instruction. The drawbacks include that identifying all the paths is a complex task and the assumptions made when simulating do not necessarily reflect reality (the ABB platform has interrupts and cache, both of which cause unpredictable behavior).
- A very accurate and still unobtrusive method is to use a logic analyzer to measure electrical signals directly from the hardware; this is one of the more work intensive and expensive strategies.
- A less expensive variant of the above is to stop the CPU at random and record what the CPU was doing. This method is useful for resource usage, since it takes samples at random in order to determine how CPU usage is spread across tasks. On the other hand it is not terribly accurate for determining actual run time. This method is popular for profiling tools.
- In their simulation of an RT database [16] set deadlines and measured the percentage of missed deadlines; in order to measure the effect of increasing the number of tasks attempting to perform a transaction every second. This is a rather low resolution method, but it answers the question of whether or not the system is viable as is.
- A sixth method is using a software analyzer that performs time measurements on the code; a software analyzer is a commercial tool that usually implements something similar to the above. Each tool has its own advantages and disadvantages [15].

A number of methods where used in this thesis, mostly for debugging purposes. Including the software analyzer called System Viewer for VxWorks. System viewer is part of Wind River workbench and shows which process is using the CPU at any given time; furthermore Wind River has a profiler which measures
how CPU time is spread between tasks and functions [17]. However the data presented based on the
stop watch method, using a timestamp register, accessed via the ABB Robotics interface.

3.4.2 Optimizing SQLite
While the short article on Stack Overflow by [18] was not peer reviewed, it gave an overview of
optimizing SQLite performance for large numbers of inserts based on some simple experiments. [18]
made a nice jumping off point giving a handful of beginner methods for optimizing SQLite. Willekes used
in-memory databases, precompiled queries and turned off indexes in order to make writes faster. One
of their fastest tests was 96000 inserts per second, or about 10 microseconds per insert, of course the
inserts where grouped into transactions which was avoided in this thesis since it was assumed that the
most time critical inserts will be not be able to wait for a large groups of data to collect.

3.4.3 SQLite with Soft Real-Time
[4] used an SQLite database to stream RFID data.
They used an in memory database with one consumer and one producer, each on their own core
communicating via an SQLite database.
In addition to reading and writing from the same database [4] also experimented with deviding the
database into several files and letting the producer and consumer task take turns accessing each file.
The consumer task was able to read from the entire database via a UNION of the tables contained on
each file. This rotating blocks method was probably to reduce the latency caused by SQLite not allowing
reads while another task writes to the database (in theory a WAL circumvents this).
The single file database was able to get an average throughput of 14ms and a worst case throughput of
20ms on an unspecified dual core processor running Linux.

3.4.4 What effects the execution time of SQLite
The primary characteristic of SQLite measured in this thesis was execution time. One facet of execution
time especially important for real-time systems is predictability. There are two factors which can
increase query time in a difficult to predict way: Compilation time for queries and time for reorganizing
the B-Tree due to both deletion and insertion, including the index B-Tree. The problem of B-Tree
balancing is problematic because rebuilding the B-Tree takes a lot of time and whether or not a query
requires a B-Tree split depends mostly on the number of previous insert queries it is beyond the scope
of this thesis to investigate the exact machinations, but the effect is clear. The question of how size of
the database affects execution time is rather interesting since the database architect has a degree of
control over the number of columns and rows in a database, and there are often scenarios where
additional rows or columns can give a non-critical but non-zero benefit to the system. In this case it can
be interesting to know what that extra row or column costs in terms of performance. It is obvious to
some degree that additional threads hurt execution time, but the exact nature is non-obvious, the
subtleties of why where discussed in depth in the SQLite section. The final factor of interest is indexes,
an additional index mean one more tree has to be written to, but they make searching easier, so how do
they affect different access types?
The effect of the following factors was considered in depth when it comes to execution time:

- Indexes
- Number of rows
- Number of columns
- Number of accessing threads
4. Method

4.1 Model
Ideally a rigorous formula that could always calculate execution time for an SQLite query given relevant data would be produced, this was not possible within the constraints of this thesis, but a series of tests may still give a general picture of how different factors affect the timing behavior of SQLite. The model is a series of test benches based on the use cases laid out in the beginning of this report, mostly the repeated querying of an SQLite database in a fashion similar to the use of CFG and Uprobe in addition to a few special cases.

Figure 5: The research process.

4.2 Experiment design – Use Cases

4.3 Use Cases
Now that the theory has been covered, the general research problem outlined in the introduction will be made into more specific use cases, these use cases will be specified again into test case questions that can be concretely answered in the method. Following which the test bench that actually carried out the tests will be described. After the results have been obtained for each test case, in the conclusion the report will go up the vine again and relate the results to the use cases. Finally the report will attempt to answer the research problem. The process is illustrated in figure 5.

This section outlines a series of use cases typical of the use of CFG and Uprobe in ABB Robotics controllers today. The use cases are not supposed to be exhaustive, merely representative for the use of SQLite in this environment. These use cases will be translated to test cases in the method. The numbering of the use cases will persist in the method and the conclusion.
4.3.1 Use Case 1, Network Configuration Retrieval
An ABB robot is usually connected to an Ethernet network when running in a field, in order to coordinate with the rest of the factory. Therefore the internet configuration has to be saved (such as IP addresses, MAC addresses, network names etc.) for internet initialization during start up. This is currently stored via the CFG database.

The following performance properties are desirable.

a. Short read time.
b. Which features are available in SQLite versus those available in CFG.

4.3.2 Use Case 2, Network Configuration Altering
As noted above the network requires configuration, of course many parts of the configuration needs to be writable by the user. This can already be a slow process since it takes a few seconds before data can be saved due to the need to charge the capacitors used as backup batteries first. So the time to write should be minimized. As above this is currently done via CFG.

The following performance properties are desirable.

a. Short write time.
b. Which features are available in SQLite versus those available in CFG.

4.3.3 Use Case 3, Real-Time Logging
When debugging the various problems on a robot it is convenient to have logging utility to store debugging messages. This sort of system should be as fast as possible since any change in timing can alter the behavior of the system being diagnosed. The current Uprobe logging tool writes to a circular buffer until a trigger makes it stop and save the last few entries. The logging tool must run for long periods of time without using large amounts of memory, hence the circular buffer. This is an important test of the real-time potential of SQLite, so predictability is important.

The following performance properties are interesting.

a. Short update time.
b. Low jitter.
c. Effect of the number of rows in database on update execution time.
d. Relative performance of SQLite and Uprobe.

4.3.4 Use Case 4, Multi-Threaded Configuration Retrieval
The controller has several Ethernet ports, and each needs information on their connection and the network they connect to, similar to case 1, but can be done in parallel. Currently network configuration is read serially, partially because CFG has poor concurrency controls.

The following performance properties are desirable.

a. Short read time.
b. Speed of parallel access.
c. Speed of serial access.

d. Effect of the number of reading threads on the above.

4.3.5 Use Case 5, Data Bus

While the other cases are at least variations on things currently done on a robot, the data bus does not currently exist in any form in the ABB system and is more presented as an interesting idea than as a necessity before switching to SQLite. The idea is a producer thread receives data from a sensor, for example heat, and records it into a database, several other consumer threads want this information, one adjusts the speed of the arm so the paint dries evenly and the other adjusts the heat of the heat gun, a third thread shares this information across the network, with the possibility of more listeners. Since this is a real time system, in order to coordinate properly, predictability is important.

The following performance properties are desirable.

a. Short of read time.
b. Speed of write.
c. Low jitter for read and write.
d. Effect of the number of reading threads.
4.4 Test Cases
This section outlines the variables in the experiments, and how they relate to the use cases. Variables where selected based on whether they were considered likely to have an effect on execution time and how similar they might be to real life use of SQLite.

4.4.1 Test Case 1, Effect of Indexes:
For the experiments on indexes a series of reads, writes and updates where performed on a pre-made database and the timing behavior was recorded, both with and without indexes on the column that the database was accessed with.

   a. On the speed and jitter of read
   Addresses use cases 1.a, 4.a, 5.a and 5.c

   b. On the speed and jitter of write
   Addresses use cases 2.a, 5.b and 5.c

   c. On the speed and jitter of update
   Addresses use cases 3.a and 3.b

4.4.2 Test Case 2, Effect of Row and Column Count
The change in execution time due to database shape should be different for different access types. The size of the database, or row count should increase search times, so the more read time increases with row count the more search time would be a component of its execution time. Update and write should be less affected since it was expected that memory related undertakings constitute the greater part of the execution time. The number of columns should increase execution time approximately linearly for almost any query.

   a. On the speed and jitter of read
   Addresses use cases 1.a, 4.a, 5.a and 5.c

   b. On the speed and jitter of write
   Addresses use cases 2.a, 5.b and 5.c

   c. On the speed and jitter of update
   Addresses use cases 3.a, 3.b, 3.c and 4.d
4.4.3 Test Case 3, Effect of the Number of Readers

In this module, a single task reading from or a single task writing to the same database was modeled. The reading task models a task reading previously stored configuration data preloaded into an in memory database. The writing task models a logging thread or configurations save. By using a single thread concurrency problems are avoided and it was possible to see a raw best case access time. Of greatest interest was: under which circumstances the best performance was achieved?

Another interesting scenario is a single task writing to the database and one or more other tasks reading from it. This test seeks to answer the question is it possible for several tasks to share a database and what are the costs? The model consists of a single task writing random information to a database while several other tasks read from the same database. The primary variable during these tests was the number of reading threads.

The central question is how much does worst case write to read time or plane read time increases with number of reading tasks. More specifically how long does it take to read information from the point writing has started? In this experiment an in memory database with a shared cache was used, as this was the only way SQLite has implemented multithreaded in memory database access.

The experiment was based around an authoritative writer task, illustrated in figure 6. The master task, here shown as write started by making sure all the other tasks where ready by reading the readersReady semaphore. After which it starts the timer and posts the testReady semaphore, signaling the read threads that they may begin reading, it then tries to start writing. When a reader was done it posted the readDone semaphore, when the writing thread has read all readDone semaphores it records the time and some other data after which it starts a new trial using the testDone semaphore (details on where the tasks get the queries they use can be found in the system architecture section). The same pattern occurred for multiple reads, however the master task read instead of wrote.
Figure 6: the testing process for multithreaded tests.
The test cases for the effect of the number of readers follows:

**a. For Data Bus**
Addresses use cases 5.a, 5.b, 5.c and 5.d

This involves an authoritarian task writing and between one and five reading tasks, using the test bench described above.

**b. For Parallel Read**
Addresses use case 4.b and 4.d

A varying number of reading threads accessed the same pre-made database, and the time between the start of the first read and end of the last read is recorded.

**c. For Sequential Read**
Addresses use case 4.c and 4.d

This experiment is essentially identical to 3.h, mentioned above, save that that the reads all occur in the same thread, in order to determine whether reading is faster when does sequentially or in parallel.

**4.4.4 Test Case 4, Write Ahead Logging (WAL)**
Addresses use cases 1.a, 2.a, 4.a, 5.a, 5.b and 5.c

According to the SQLite documentation a WAL can make SQLite slightly faster, particularly writes [10]. So a number of tests were performed using a write ahead journal and the default delete journal for read and write.

**4.4.5 Test Case 5, Replacing the backend of Uprobe**
Addresses use case 3.d.

The real-time logger Uprobe must be able to record debug data as quickly as possible. As a test of the real-time worthiness of SQLite the backend of Uprobe was replaced by one based on SQLite. Since Uprobe allocates its memory beforehand it was considered acceptable to create the entries to be updated beforehand, in order to improve performance.

Since CFG is mostly for storing configuration data for startup it has no real-time requirements other than it “be responsive” and the advantages of features such as searching and relations gives SQLite a clear advantage. However can SQLite be used for real-time processes? CFG is not used in any real-time process, Nevertheless the usefulness of SQLite for real-time tasks is still of interest due to its many features. The most obvious candidate system for the tests is the real-time logging system Uprobe.

Uprobe uses a circular buffer to store messages under runtime. It saves pointers to data that the client program gives it, and then when the test is over Uprobe follows the pointers and saves the information into an output file.
A new version of Uprobe was made where the backend was redone with SQLite. The SQLite version saves the exact same data as Uprobe would and then saves the database to hard disk so it can be compared to Uprobe output. The output between the two versions was not identical since SQLite was initialized to a certain number of entries while a circular buffer is initialized to a certain amount of data in bytes, so the circular buffers end up cutting of at different points.

In order to compare the performance of the two versions a test bench ran a large number of logs through Uprobe, and the timing was compared.

To summarize, below are the test cases and the use cases. The columns are test cases and the rows are use cases:

<table>
<thead>
<tr>
<th>Use Cases</th>
<th>tc1a</th>
<th>tc1b</th>
<th>tc2a</th>
<th>tc2b</th>
<th>tc3a</th>
<th>tc3b</th>
<th>tc3c</th>
<th>tc3d</th>
<th>tc4a</th>
<th>tc4b</th>
<th>tc4c</th>
<th>tc4d</th>
<th>tc5a</th>
<th>tc5b</th>
<th>tc5c</th>
<th>tc5d</th>
</tr>
</thead>
<tbody>
<tr>
<td>uc1a</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uc1b</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uc1c</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uc2a</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uc2b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uc2c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>uc3a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>uc3b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>uc3c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uc4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uc5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Coverage of the use cases by test cases

Assumptions

In the experiments here discussed there were certain assumptions, including that the interaction between the database tasks and other tasks obfuscates rather than makes clear the underlying factors. Additionally it is assumed that data access patterns can be approximated by random distributions. Moreover it is assumed that the overhead caused by the measurements do not unduly change the nature of the timing.

4.5 System architecture

4.5.1 Hardware

The hardware used in testing consisted of an ABB robot controller which is based on the Intel Atom D425 processor with external RAM and a hard drive consisting of an SD card.

An interesting feature of the robot controller is that when it loses external power it retains enough charge in a bank of capacitors with which to do a few last second tear down tasks. A special computer process, moves a specified region of the RAM memory to the SD card. This means that an in memory database can be as persistent as one on hard disk (However this is only possible if the in memory database is on a RAM file system which involves more overhead than a simple memory databases). This
means that the classic drawback of an in memory database, that it disappears if you lose power, is less of an issue for the robot controller.

4.5.2 Software
The general system for creating a database and determining the queries to be tested, was as follows (see the diagram in figure 7): first an experiment profile which specifies the number of rows, columns and queries was written. Then a SQL generating script written in Python uses this profile to generate a list of queries and other directives to generate an experiment “plan”. After that the plan was transferred to the test bench on the robot controller which generates a database to be tested against. Then the test bench compiles the queries now the test bench has entered the main loop where it first binds data then it starts a timer and executes the precompiled query against the test database. When it is done it stops the time and records statistics in a data log. The test bench continues to iterate through experiments until it has gotten through the entire plan.
After the data has been logged in a data log implemented as an SQLite database it is transferred to a PC for analysis. The data was mostly analyzed using Python, and all the graphs produced where made using the Python library Matplotlib [19].
In the tests performed, access was for the most part random, both because it is easier and because it is most like worst case behavior (since the data is unlikely to be cached); excepting multithreaded reading which may be slower when accessing what was being written by another task (since it has to wait for locks).

When possible, the tests where run at high priority and the scheduler was shut off, if the timer interrupt was triggered during the test the test was redone (since it was not possible to deactivate the timer interrupt). This was to assure that the times measured were the reading or writing task alone. The problem of compilation time was circumvented by binding the values specific to a test query to a precompiled query, this is a standard method for increasing performance in SQLite. No solution to B-Tree splitting was found, save not inserting anything into the database, see the experiments on write and update. This was the method used in Uprobe logging test.
5  Result and Discussion

5.1 Test Case 1, Effect of Indexes:
Single threaded experiments model a single reading task and a single writing task repeatedly accessing the same database. The single reading task reads repeatedly from a preloaded database. The single writing task repeatedly writes strings to a database. The test task runs at high priority and with the scheduler is shut off. The first series of experiments will look at jitter, the same list of queries is carried out three times and the difference in behavior is noted, in addition the effect of whether or not the test is indexed is tested.

5.1.1 Writing, With and Without Index
First shown is write without an index, see below, note how much closer the second two trials are compared to the first, also the three peaks which were also found in indexed write, in figure 8 is a distribution graph is a selection of execution times of the trials, it shows how often different execution times occurred, it ended up being a bell curve:

<table>
<thead>
<tr>
<th>Average run time</th>
<th>Maximum run time</th>
<th>Maximum jitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>44,6 µs</td>
<td>58 µs</td>
<td>20 µs</td>
</tr>
</tbody>
</table>

Table 2: Execution time of unindexed write.

![Distribution, Unindexed writing](image)

Figure 8: Distribution of unindexed write.
The peak furthest to the right may be due to table splitting, especially since trials taking longer than 50 microseconds tend to appear approximately every 12 or 24 tests. What the peak furthest to the left represents is unclear.

In figure 9 are the results for indexed write, note that the graph looks like a slightly slower version of unindexed write, save for an additional peak far to the right, at about 125 microseconds. Runtime went up by 22 microseconds max run time went up by 83 and maximum jitter by 50.

<table>
<thead>
<tr>
<th>Average run time</th>
<th>Maximum run time</th>
<th>Maximum jitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 µs</td>
<td>141 µs</td>
<td>90 µs</td>
</tr>
</tbody>
</table>

Table 3: Execution time of indexed writing.

![Distribution, Indexed writing](image)

Figure 9: Distribution of indexed write.

As close as the different trials are to each other, it can be seen in the sample bellow that the trials are not copies of each other, in addition there seems to be 4 “classes” of inserts that correspond to the peaks of the distribution graph:

The indexed writing tests using identical queries and databases where very close to each other. It is possible that since the set of queries are identical and the beginning database was the same that the
timing profile was due to the changing shape of the database, so are the trials as closely distributed when accessing different queries and databases? To test this, a series of writes with different databases and query series was performed (indexed). Since reads do not change the shape of the database, the same experiment is not interesting for reads. Figure 10 is a graph of trials from three different “plans” side by side. Though maximum run time and maximum jitter went up a bit the two trials are rather simular.

<table>
<thead>
<tr>
<th></th>
<th>Average run time</th>
<th>Maximum run time</th>
<th>Maximum jitter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>65,8 µs</td>
<td>158 µs</td>
<td>108 µs</td>
</tr>
</tbody>
</table>

*Table 4: Execution time of indexed writing with separate plans*

![Distribution, for indexed writing: separate plans](image)

*Figure 10: Distribution of indexed writing with separate plans.*

### 5.1.2 Reading, With and Without Index

The next experiment was indexed reading, which was generally faster than writing. The jitter was comparable to the other tests. As can be seen in figure 11, the majority of queries take approximately 41 microseconds, excepting the group at 36 microseconds. Jitter was about 16 microseconds.

<table>
<thead>
<tr>
<th></th>
<th>Average run time</th>
<th>Maximum run time</th>
<th>Maximum Jitter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>40,7 µs</td>
<td>50 µs</td>
<td>16 µs</td>
</tr>
</tbody>
</table>

*Table 5: Execution time of indexed reading.*
Figure 11: Distribution of indexed read.

For the next experiment reads without an index where performed. They were significantly slower. Run time went up 222 microseconds, maximum run time up by 233 and jitter by 12. In addition, as can be seen in figure 12 the reads where divided into two equally sized peaks instead of the large and the small one seen in indexed reading.

<table>
<thead>
<tr>
<th></th>
<th>Average run time</th>
<th>Maximum run time</th>
<th>Maximum jitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>trial</td>
<td>262.4 µs</td>
<td>283 µs</td>
<td>28 µs</td>
</tr>
</tbody>
</table>

Table 6: Execution time of unindexed read.
The two peaks can be seen in the indexed reading, though why the faster peak grows so much is unclear, as a matter of fact it is unclear what the two populations are to begin with. The high and low execution times do not seem to align query by query, though the groups are clearly present in all trials.

Though for very write intensive systems indexing may not be worth the benefit to reading compared to its cost to writing it nevertheless in most cases the benefit is clear.

5.1.3 Update, With and Without Index
Write was problematic since it inserts a new entry into the table and can require rebalancing the tree. If a predetermined number of entries where to be written the entire database can be created beforehand and the entries merely updated, which should take less time. However this was not what was observed, perhaps an update accessing the database via a primary key would be faster. The execution times are graphed in figure 13, jitter on the order of 15 microseconds:

<table>
<thead>
<tr>
<th>Average run time</th>
<th>Maximum run time</th>
<th>Maximum jitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>67,2 µs</td>
<td>87 µs</td>
<td>24 µs</td>
</tr>
</tbody>
</table>

Table 7: Execution time of indexed update.
Finally is unindexed update, graphed in figure 14. The average and worse case execution time are far better than write, update without an index on the other hand is far worse than even indexed writing and should probably be avoided, Average run time increased by 248,3 microseconds and maximum runtime by 256 maximum jitter by 4:

<table>
<thead>
<tr>
<th>Average run time</th>
<th>Maximum run time</th>
<th>Maximum jitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>315,5 µs</td>
<td>343 µs</td>
<td>28 µs</td>
</tr>
</tbody>
</table>

Table 8: Execution time of unindexed update.
A summary of the results follows:

<table>
<thead>
<tr>
<th></th>
<th>Average run time</th>
<th>Maximum run time</th>
<th>Maximum jitter</th>
<th>Max jitter/Average run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unindexed write</td>
<td>44.6 µs</td>
<td>58 µs</td>
<td>20 µs</td>
<td>0.45</td>
</tr>
<tr>
<td>Indexed write</td>
<td>66 µs</td>
<td>141 µs</td>
<td>90 µs</td>
<td>1.36</td>
</tr>
<tr>
<td><strong>Difference</strong></td>
<td><strong>-84%</strong></td>
<td><strong>-82%</strong></td>
<td><strong>-84%</strong></td>
<td>+0.91</td>
</tr>
<tr>
<td>Unindexed read</td>
<td>262.4 µs</td>
<td>283 µs</td>
<td>28 µs</td>
<td>0.11</td>
</tr>
<tr>
<td>Indexed read</td>
<td>40.7 µs</td>
<td>50 µs</td>
<td>16 µs</td>
<td>0.39</td>
</tr>
<tr>
<td><strong>Difference</strong></td>
<td><strong>+48%</strong></td>
<td><strong>+140%</strong></td>
<td><strong>+350%</strong></td>
<td>-0.28</td>
</tr>
<tr>
<td>Unindexed update</td>
<td>315.5 µs</td>
<td>343 µs</td>
<td>28 µs</td>
<td>0.09</td>
</tr>
<tr>
<td>Indexed update</td>
<td>67.2 µs</td>
<td>87 µs</td>
<td>24 µs</td>
<td>0.36</td>
</tr>
<tr>
<td><strong>Difference</strong></td>
<td><strong>-79%</strong></td>
<td><strong>-75%</strong></td>
<td><strong>-14%</strong></td>
<td>-0.27</td>
</tr>
</tbody>
</table>

Table 9: Summary of execution times of various queries under different circumstances

As expected indexes made read faster and write slower. The data shows that if read and write were used equally often average run time was decreased by 100 microseconds, maximum run time by 75 however average maximum jitter would increase by an average 29 microseconds. The breakeven point
is when write is used 73.7 times for every 26.3 times read is used (about 3 writes per read). If update and read are used together average run time goes up by 235, maximum run time by 244.5.

Longer run times mean more background noise which in turn widens the run time distribution. In order to counteract this the jitter has been normalized in order to make relative jitteriness more obvious, the clear winner being write, especially indexed write, though indexes in general increase relative jitter. A significant portion of the jitter can probably be attributed to cache misses and hardware interrupts which would be impossible to get rid of without using drastically different hardware (disabling the cache was attempted but the program became extremely slow allowing more hardware interrupts so that the gains in predictability due to cache loses was entirely covered up). In addition the imprecision of the timer gives an apparent jitter of a few microseconds. Write probably has the most jitter due to rebalances in the tree or memory allocation when its current pool has run out.

5.2 Test Case 2, Effect of Row and Column Count
Next, the effect of the number of columns and rows on execution time will be investigated; in the following tests the same data is presented twice, once organized by number of columns and again by number of rows. In order to show their respective effect on execution time.

5.2.1 On the speed and jitter of read
First the results for varying the number of rows on jitter in read, shown in figure 15, below.

![Distribution of read with varying number of rows](image)

**Figure 15:** Distribution of read with varying number of rows.
And then the relationship to execution time for average and worst case, shown in figure 16 and 17 below:

**Figure 16: Execution time vs. row count average case: read.**

**Figure 17: Execution time vs. row count worst case: read.**
The average case shows something like a logarithmic scale. However the worst case was so noisy no real conclusion can be drawn. Since worst case is the longest execution time for a set of trials the worst cases for read are due to some extraordinary event not controlled for. At any rate the row count has no strong correlation to worst case read execution time.

Next the number of column is considered, it has a greater effect than increasing the number of rows (at least in the ranges tested here). As noted before however the use of an index or B-Tree means increases in execution time due to larger databases is less pronounced and runtime becomes dominated by overhead. Note that the execution time versus number of rows for reads without an index was far more marked, (see appendix C: Rows effect on unindexed execution time).

Then there is the effect of the number of columns has on reading, shown below in figure 18 there was a very clear increase of a few microseconds for each two columns 0,2 microseconds per column average and 0,2 for worst case.

![Figure 18: Distribution of read with varying number of columns.](image-url)
As can be seen in figure 19 and 20 above, increase in average time versus column count is clear and looks linear, however the shape of the worst case was unexpected; if the frequency chart is to be believed the slow 5 column execution time is an outlier that happened to occur during that test. This
od outlier is definitely interesting but investigating it is beyond the scope of this thesis. Otherwise the increase in worst case execution time seems to be vaguely linear.

5.2.2 On the speed and jitter of write

Below in figure 21 is a graph of the effect of the size of the database on write:

![Figure 21: Distribution of write with varying number of rows.](image)

The difference was not nearly as marked as for columns. In part because there was so much more noise, but the average run time goes up steadily for both average per and worst case, though slightly sporadically.
As can be seen in figure 22 and 23 above, the increase in execution time seems to almost logarithmic, though obviously further research is required. Theoretically of course it makes sense additional columns always require an additional write, giving a linear increase. However all write trials showed a dip in
execution time around 700 rows, the reason behind this is unclear perhaps a database of that size requires another page of memory and for some reason a two page system is most efficient. In addition unlike reads writes have a significant overhead in memory allocation which can cover any search times. The increase in execution time for additional rows was no more marked without the index, (see appendix C: Rows effect on unindexed execution time).

Below in figure 24 are all the write tests organized by number of columns:

![Distribution, number of columns written](image)

**Figure 24:** Distribution of write with varying number of columns.
Figure 25: Execution time vs. column count average case write.

Figure 26: Execution time vs. column count worst case write.

It can be observed in figures 25 and 26 above that the relationship between number of columns and average and worst case execution time is clear, and vaguely exponential.
5.2.3 On the speed and jitter of update
Due to the similarities between update and read only the average case of update is shown. First is the effect of row count on update with index, figure 27. In figure 28, execution time goes up with row count, a dip similar to the one seen in write is also present, if they are related, perhaps it is due to some kind of caching.

![Distribution of update with varying number of rows](image)

Figure 27: Distribution of update with varying number of rows.
Figure 28: Execution time vs. row count average case update.

Then the effect of the number of columns written to for update in figure 29:

Figure 29: Distribution of update with varying number of columns.
Followed by the effect on average case execution time in figure 30:

![Graph showing execution time vs column count, average case: update.](image)

Figure 30: Execution time vs. row count average case update.

To summarize, the row data will be the focus of the analysis since a database generally has far more rows than columns:

From the graphs above the effect of column count on write seems to be mostly linear and row count seems to have vaguely logarithmic effect. The same appears to be true for read, though worst case appears to have too much noise to draw any clear conclusions. Of course these conclusions only hold true with an index, see appendix B and C for details, consider however that write without index is not massively affected by row count read without index becomes approximately linearly slower with increased row count.

In it can be concluded that column count has a greater effect, of course in practice column count only rarely increases as much as row count.

### 5.3 Test Case 3, Effect of the Number of Readers

This section is an attempt to better understand the behavior of SQLite when there are several threads interacting with the database. The main question to answer was how the time it takes to write and read increase with number of reading tasks.

In this experiment an in memory database with a shared cache was used, this is the only way SQLite has implemented multithreaded in memory database access and actually enables more levels of locking. This required upgrading the SQLite on ABBs VxWorks controller to the latest version. The tests will concentrate on 256 rows and 9 columns, a sort of mid-range for the cases discussed in this thesis.
The method utilized was an authoritative main task that, after making sure all the other tasks were ready, starts writing and only after that allows the readers to read the data it posted. Finally, semaphores where in place to allow record keeping between measurements.

5.3.1 For Data Bus
One of the most interesting questions is how the number of reading tasks changes execution time. This is modeled by a single task writing random information to a database while several other tasks read from the database, the results are shown in figure 31:

![Distribution, number of reading threads](image)

**Figure 31:** Distribution of execution time for reading threads for data bus with varying numbers of readers.

The execution time increases rapidly with the number of threads, the average increase in execution time is on the order of 30 microseconds. Below is a table of average runtime, max runtime and jitter in relation to number of readers. The change in maximum run time is shown below

<table>
<thead>
<tr>
<th></th>
<th>1 Reader</th>
<th>2 Readers</th>
<th>3 Readers</th>
<th>4 Readers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max runtime(µs)</td>
<td>242</td>
<td>258</td>
<td>300</td>
<td>341</td>
</tr>
<tr>
<td>Average runtime(µs)</td>
<td>141.0</td>
<td>181.0</td>
<td>223.0</td>
<td>262.8</td>
</tr>
<tr>
<td>Max jitter(µs)</td>
<td>127</td>
<td>98</td>
<td>99</td>
<td>101</td>
</tr>
</tbody>
</table>

**Table 10:** Execution time and jitter in relation to number of reading threads in a databus configuration.

For some reason the jump from one to two reading threads is more efficient than just adding an additional read (remember average read time is about 40 µs), but otherwise the time increase is about the same as reading serially (at least with up to 4 threads). There doesn’t seem to be a strong benefit for multithreading, but no real penalty either.
5.3.2 For Parallel Read

In this section several threads read from the same database without any writing thread, in order to determine whether several tasks reading from a database was any faster than doing it in one thread, results in figure 32:

![Distribution, number of reading threads](image)

**Figure 32:** Distribution of execution time for reading threads for parallel read with varying numbers of readers.

Below is a table over the specifics:

<table>
<thead>
<tr>
<th>Max runtime(μs)</th>
<th>1 Reader</th>
<th>2 Readers</th>
<th>3 Readers</th>
<th>4 Readers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average runtime(μs)</td>
<td>43.2</td>
<td>103.0</td>
<td>143.3</td>
<td>183.7</td>
</tr>
<tr>
<td>Max jitter(μs)</td>
<td>16</td>
<td>28</td>
<td>20</td>
<td>38</td>
</tr>
</tbody>
</table>

Table 11: Execution time and jitter in relation to number of reading threads in a parallel read configuration.

**a. For Serial Read**

For comparison, below is the same test, but carried out on the same thread, figure 33:
Figure 33: Distribution of execution time for reading threads for serial read with varying numbers of readers

It appears that sequential access is faster, and that the overhead of using several threads is greater than any potential benefits, at least on a single threaded system:

<table>
<thead>
<tr>
<th></th>
<th>1 Reader</th>
<th>2 Readers</th>
<th>3 Readers</th>
<th>4 Readers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max runtime(µs)</td>
<td>58</td>
<td>100</td>
<td>126</td>
<td>161</td>
</tr>
<tr>
<td>Average runtime(µs)</td>
<td>48.7</td>
<td>79.0</td>
<td>110.9</td>
<td>145.8</td>
</tr>
<tr>
<td>Max jitter(µs)</td>
<td>14</td>
<td>25</td>
<td>19</td>
<td>21</td>
</tr>
</tbody>
</table>

Table 12: Execution time and jitter in relation to number of reading threads in a serial read configuration.

5.4 Test Case 4, Write Ahead Logging (WAL)
Recall that Write ahead logging enables SQLite to read while writing, while this is useful in some cases it can come at a performance cost, but how much of one? In this section timing tests where done with and without a WAL.

The difference between tests was subtle so here is a summary of the numerical data (for distribution graphs see Appendix A: Graphs for WAL):

<table>
<thead>
<tr>
<th></th>
<th>Minimum runtime</th>
<th>Maximum runtime</th>
<th>Average runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete mode</td>
<td>33</td>
<td>64</td>
<td>53</td>
</tr>
<tr>
<td>WAL mode</td>
<td>34</td>
<td>63</td>
<td>54</td>
</tr>
</tbody>
</table>

Table 13: Runtime with and without WAL.

There was a minor increase in execution time, though only about a microsecond.
The same experiment was carried out for writes (delete versus WAL) and the net effect was to be an improvement on the order of five microseconds in the worst case (again, for distribution graphs see Appendix A: Graphs for WAL), therefore WAL mode will be using it for the remaining trials.

5.5 Test Case 5, Replacing the backend of Uprobe

Uprobe is ABB robotics real-time logging utility. Below is a graph of trials comparing SQLite and a circular buffer as a backend in a simplistic log saving test bench. SQLite used update against an indexed database, The index column of the database was not altered by update since this could cause tree rebalancing and so forth. It would be interesting to try by accessing via a primary key, or using write which seems to be faster (of course the ever increasing database size would eventually make that problematic), however neither of these had an execution time in the range of the circular buffer which only took 1-2 microseconds its line is barely visible at the bottom of the graph in figure 34 below while SQLite jumps up and down between 50 and 100 microseconds.

![Figure 34: execution time of Uprobe with and without SQLite](image)

SQLite was clearly slower than the circular buffer it replaces, which was to be expected since SQLite has significant overhead even when used as an in memory database, in applications where timing is critical SQLite is probably not the right choice. It may be worthwhile to temporarily save data in a circular buffer and then save the logs onto the hard disk as SQLite for ease of access and portability, or in to use SQLite in less time critical applications like saving configuration data.
5 Conclusion

6.1 Result Summary

TC1. Effect of indexes

For single threaded Read and write, when there is an index read is faster than update which is faster than write. Without an index write is far faster than read which is faster than update. Without an index the performance of read and update is abysmal. When writes are three times more common than read an index decreases performance. For some reason a database with about 700 rows seemed to be the most efficient, further work is required to determine why. Perhaps this was the threshold at which SQLite adds another page of memory and this introduces some kind of efficiency.

TC2. Effect of Row and Column Count

Increasing the number of columns and increasing the number of rows increases execution time. The most dramatic changes are for read and update without an index, in that kind of database the number of rows are of great importance. Otherwise the increases were relatively similar: decreased database size decreases the number of queries.

TC3. Effect of the Number of Readers

Adding one additional reading thread is more efficient than reading from the same thread, but otherwise the execution time seems to increase at about the single threaded rate.

TC4. Write Ahead Logging (WAL)

WAL was shown to be slightly more effective in the tests performed, though not by much.

TC5. Replacing the Backend of Uprobe

The Uprobe test did little more than demonstrate that SQLite, while fairly fast, is no comparison to raw data access and when speed is more important than searching time and feature count; SQLite is probably not the right choice.

Many of the results where unsurprising, reading was faster than writing. Indexing made writing slower but updating and reading faster, nevertheless the tradeoff is worth it unless a disproportionate number of writes is performed. Background jitter is clearly an issue but is at least partially due to the platform itself.

6.2 Limitations

These results are not a set of laws giving worst case and average execution times for all imaginable cases. Instead this thesis is intended to give a general picture of what kind of data architectures and settings can increase SQLite performance, most of the tests where done with in WAL mode, on a single platform and with an in memory database. Furthermore the benchmarks are randomly generated
queries and may not reflect real life configurations. For other applications where the number of rows and columns or the access pattern is more precisely known, new tests may be in order.

6.3 Implications

The purpose of this thesis was to first to assess the usefulness of SQLite for use in CFG and Uprobe and second to create a set of guidelines for the use of SQLite in an ABB robot controller in a way that optimizes performance. It is beyond the scope of this thesis to derive a formula to predict execution time for SQLite, but the following rules of thumb can be derived from the results in order to minimize SQLite execution times, each guideline group is coupled with a use case, (see the Background):

UC1. Network Configuration retrieval
   a. The results show using an index decreases average and worst case read time significantly.
   b. Keep the number of columns and rows to a minimum, it was demonstrated that execution time increases with both row and column count.

UC2. Network Configuration Altering
   a. When read-time is less important indexes should be avoided, since results show indexes increase write time more than they decrease read time.
   b. The results show that in general smaller databases are faster, but around 700 rows seems to be optimal. Execution time also increases with column count so it is advisable to avoid unnecessary columns.
   c. Average execution time of write is faster than update, but worst case update is slower than writes.
   d. Each write increases the database row count, so if entries are never deleted update will eventually become faster than write.

UC3. Real-Time Logging
   a. Since indexes help update speed so much indexes should be used, but writing to the indexed entry will destroy these gains.
   b. Again, smaller databases are faster.
   c. SQLite logger is significantly slower than a non-database solution.

UC4. Multi-Threaded Configuration Retrieval
   a. Serial access is faster, so avoid multithreaded access if possible.
   b. In addition increasing the number of threads increases execution time linearly, multithreaded is most viable for two threads, and only gets worse in comparison to sequential access after that.

UC5. Data-Bus
   a. The combination of two threads reading threads and one writing thread was the most efficient, but the general slowness of the process means the idea might not be viable in the scenario brought forth and mailboxes are probably a better idea for such architectures, particularly since there is only one writer and the risk for collisions is low.

In addition here are a few general tips when using an SQLite databases:
Indexes make read, update a lot faster and writes a bit slower, this effect is magnified by the size of the database, reads and updates gain relative performance while write loses its relative performance as the database grows. All execution times increase with the number of rows and columns, but since the row count is more likely to grow, their number must be more carefully considered, though the effect is a close to logarithmic increase in execution time. A database where reads are the more common than writes or updates and an index on the queried column is very fast. However, if writes happen more than three times as often as reads indexes should be avoided. Update is something of a specialized tool, and is only faster than write when comparing worst case and only then when used with indexes. On the other hand for very large, indexed databases; update should to be faster than write. Though SQLite is clearly not suitable for applications with extremely short deadlines like Uprobe, it is easy to use and useful for tasks with deadlines ranging between 50 to 400 microseconds (or longer), the acceptable deadline depending on whether the structure of the database and whether it is being read from or written to. That is not to say SQLite will always lose on speed. For very specialized applications where large databases must be searched quickly and often, SQLite will win over simpler data structures using linear searches. In fact in most applications when the searching, indexing or other more unusual features can be taken advantage of SQLite will probably win out. When an environment has lax deadlines SQLite is often the superior choice due to its large number of features, even on lean embedded systems such as the robot controller.

Perhaps most importantly this thesis quantifies the advantages of different configurations and strategies for using SQLite, which is important for using SQLite on a deadline but also for making it “responsive” in configuration loading and saving applications such as using SQLite as a replacement for CFG. In conclusion, the performance of configuration saving and loading can be improved substantially; however the use of SQLite as a real-time tool is not viable in the robot controller as with the state of technology today.
6.4 Future work

6.4.1 More Specialized Tests
Now that an overview study has been performed, if a specific application is of interest a specific number of columns threads and rows could be more thoroughly tested. In addition it would be interesting to see how very large databases fare (perhaps up to 100,000 rows), especially in write intensive environments (writes suffer a decrease in execution time from indexes, but is the trend maintained forever?).

6.4.2 Mathematical Model of Execution Time
It would be fascinating to make a mathematical model of the database in order to determine execution time. How close such a model is to reality probably dependent on the amount of effort spent on it (cache misses, interrupts and timing resolution probably give an upper bound for accuracy), but it would be nice to be able to predict different “humps” on the graph occur. It would be interesting to see the upper bound of the execution time for a query (queries could be made more efficient, perhaps something could be done if it is known a query will not make its deadline). A first step to creating such a system would be to make a simple model of the B-Tree since the fact that it splits is a large contributor to execution times. It would be most practical to construct such a model for a specialized case (number of rows and columns known).

6.4.3 How Fast is Responsive?
As stated in the introduction an important feature of the transfer between CFG and SQLite is how responsive it is, however what “responsive” entails is never really quantified. It would be interesting to design an experiment to determine at what point a program stops being perceived as responsive and starts being perceived as sluggish.

6.4.4 R-Tree Collision Detection
SQLite has a feature called an R-Tree tree which is a special type of index on a numerical entry that used in mobile phones to quickly determine which GPS points to display on screen. Since robots move in a 3-D space there may be some navigational applications which can take advantage of this. Collision detection is one such potential application, perhaps in order to allow for navigation of an environment mapped with a laser scanner. There are probably others which are viable, for example for target selection.
Works Cited


[Accessed 29 March 2013].


Appendixes, Additional Experiments and Graphs

Appendix A: Graphs for WAL

A.1 Read
Here are the graphs for execution time of reading with journal mode equal to DELETE (the default form), the serial numbers are merely the data and exact time.

Figure 35: Distribution reading without WAL

Then the exact same queries, this time with a write ahead log, which the SQLite documentation says if faster for most write cases but slower for reads:
Figure 36: Distribution reading with WAL

A.2 Write
The same experiment was carried out for writes, first in delete mode:

Figure 37: Distribution writing without WAL

Then in WAL mode:
The net effect seems to be a slight improvement in the worst case.
Appendix B: Row, Column Counts Effect on Update Execution Time

B.1 Row count
First is the effect of row count on update with index, execution time goes up, but only by 0.3 microseconds per additional row on average and 0.5 for worst case:

![Distribution, number of rows in database](image)

Figure 39: Execution time vs. row count indexed update.

Below is the effect of row count on update time without an index, much more marked, 2.3 us/row average and 4.6 worst case. Recall that, as noted in the result section, update without an index is crippling slow:
Figure 40: Execution time vs. row count unindexed update.

**B.2 Column count**

First with index: 0.3 us/column average and 0.5 maximum:
Figure 41: Execution time vs. column count indexed update.

Then without, note that this graph had to be clipped so that the trend would be more visible, as noted above unindexed update quickly becomes very slow. Other than that however it follows the general tread on clear increase in execution time 1.8 us/column average and 4.6 worst case:
Figure 42: Execution time vs. column count unindexed update.
Appendix C: Rows Effect on Unindexed Execution Time

C.1 Write
Execution time versus number of rows for written without index, 0.26 us/row average and 0.4 worst case:

Figure 43: Execution time vs. column count unindexed write.
Note that the main difference distribution is not markedly different from the results in with an index save that it is about 10 microseconds faster and lacks a hill at 125 microseconds (see Number of Columns and rows under Results)

**C.2 Read**

Execution time versus number of rows for read without index:
Figure 46: Execution time vs. column count unindexed read.

Figure 47: Execution time vs. row count average case unindexed read.
As can clearly be seen the increase in execution time is linearly proportional for both average and worst case execution time. In addition, as can be seen from the graphs above, average and worst case are very close to each other.
Appendix D: Different Memory Types

Below is a comparison in speed between different memory types, starting with hard disk, that is to say against the SD card, continuing to a file system on the RAM and finishing with a true in memory database, with execution time clearly improving for each step. Below is the performance for each media type:

<table>
<thead>
<tr>
<th></th>
<th>Average (µs)</th>
<th>Worst Case(µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD kort</td>
<td>26.338</td>
<td>153.285</td>
</tr>
<tr>
<td>RAM disk</td>
<td>937</td>
<td>1278</td>
</tr>
<tr>
<td>In memory</td>
<td>74</td>
<td>149</td>
</tr>
</tbody>
</table>

Below are graphs for the jitter:

![Figure 49: Distribution for hard drive execution time write.](image)
Figure 50: Distribution for Ram disk execution time write.
Figure 51: Distribution for in memory write.