GOAS: A compiler driver for algebraic queries

Josh Reese

MASTER OF SCIENCE

COMPUTER SCIENCE

SCHOOL OF INFORMATICS

UNIVERSITY OF EDINBURGH

2012

Acknowledgements

I would like to sincerely thank my supervisor, Stratis Viglas, whose guidance and assistance have prevented me from floundering and sinking.

Additionally, I would like to thank Andi Hellmund from the gcc mailing list for helping me waste less time on fruitless endeavors.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

()

Table of Contents

Chapter 1

Introduction

Some material for this thesis can also be found in [14].

It is the case that many database servers in operation today only run a small set of different queries. Each time these queries are executed they must be reinterpreted by the system. This behavior can cause a database server to waste large amounts of time redundantly interpreting the same query. In addition to this when a query is interpreted the underlying hardware isn't taken into consideration. Without considering the hardware that is executing these queries modern database systems cannot operate at their maximum potential.

This document presents two items: an algebraic language that can be used to evaluate queries, called oas, and a compiler for that language, called goas. This language consists a series of algebraic operations (e.g. scan, select, and project) while the compiler produces a machine specific executable which will carry out those operations. The concept of compiling database queries stems from recent work in the area of optimizing queries for modern hardware.

Some benefits to the approach of query compilation include generating machine specific executable queries and generating queries that use CPU cache and registers more effectively than interpreted structured query languages (SQL). In the past researchers have tried techniques such as decompressing tuples prior to iteration, producing multiple tuples on operator calls, producing all the tuples for a query at once, and compiling the query into Java Bytecode [11]. Recent approaches have been taken to examine this issue by using code templates to generate compiled queries, translating queries to pseudo code which is then translated to Low Level Virtual Machine (LLVM) assembly, and using vectorization [9, 15, 11]. One fact that is common among these projects is that a query will run faster once compiled. However, there is a non-

negligible overhead in the time it takes to compile a query. It is for this reason the goal of this project is to create the framework for a library of executables that will only need to be compiled once but will be run many times. By creating this library database servers will be able to process more queries in less time thus increasing their overall performance.

1.1 Motivation

Due to the nature of database queries (i.e. no advanced knowledge about the query at runtime) most systems are required to use some form of an interpretation engine [15]. This is known as the iterator model and was designed to limit disk I/O. Today it is likely that large database servers will often be able to fit all of their information into main memory [9]. Since this model, and many of the algorithms used in these queries, were designed to limit disk I/O they are no longer the best choices for all query engines.

There is a large amount of bloat associated with modern query engines that may not be necessary for all applications. For example in an embedded database that is running the same set of queries repeatedly it isn't really necessary to have a query optimizer. The optimized query could be obtained elsewhere and that could be embedded into the system. These queries that are being run over and over again could be compiled into executables that the embedded system could run immediately and thus bypass almost all query processing altogether. With these machine specific executables a system will be able to execute queries much faster, thus improving the overall performance. In addition to this if a relation can fit entirely in main memory it isn't necessary to have the amount of function calls a modern query engine will use to work with a query. These function calls are very expensive and if a system could reduce the amount of calls it needs to make, once again, it could improve the overall performance. One way to do this would be to load the relation into main memory and access it via standard reads and writes. On top of this modern query engines pay a large cost to manage transaction processing. But this isn't always needed so the cost becomes an unnecessary overhead.

Another very interesting aspect of compiling queries for specific machines is the opportunity for various optimizations. For example if code is known to be running on a multi-core machine various parallelizations could be introduced into the code. In addition to this there are various optimizations available in modern compilers that could be applied to the queries as they are compiled.

Chapter 2

Current Query Engine Model

Most current query engines are based on the iterator model [7]. Queries are interpreted at runtime and broken into various algebraic operators. These operators implement an application programming interface (API) that consists of functions *open()*, *next()*, and *close()*. Where open() will initialize the state of the operator (setting up memory for I/O buffers and pass in arguments the operator may need, such as selection conditions), each call to next() produces a new tuple for the operator, and close() deallocates what open() allocated [12]. This technique, because it is made to be completely generic, exhibits a large number of function calls for each tuple that needs to be propagated through the query plan. The model described here is not entirely conducive to efficient execution on modern processors. Each of these function calls will need to either save or restore the contents of the processors registers to or from the stack which means the processor will not be able to use its registers effectively [9]. With registers being the fastest data access available to a modern processor this is not an ideal situation. Not only will processors not be able to use their registers effectively but since each function call is a branch in execution this forces a new instruction stream to be entered into the pipeline which limits the superscalar execution inherent in modern processors [9].

2.1 Operators

There are a number of operators available in a query engine. This project focuses only on the unary operators: scan, select, project, and sort. The purpose of the scan operator is to produce the tuples present in a relation for other operators to manipulate. The select operator evaluates individual tuples based on a predicate, or series of predicates, and removes tuples from the output relation that don't satisfy the predicate(s). Project

is given a list of keys and will remove any column from a relation which is not one of those keys. Finally, sorting takes a set of keys and sorts the output relation, in some predetermined order, according to the values associated with each key.

2.2 Operators and the Iterator Model

When a query has been broken into a relational algebraic form it is then passed to the query optimizer which will choose specific operators for each of the algebraic operations (e.g. nested loops join) and form a plan for executing the query. These operators communicate with each other via iterators in this model. In addition to this a query optimizer will also have to determine the order in which these operators connect to one another. There are two types of plans: deep plans and bushy plans with deep plans being more suited to pipelining and bushy plans more suited to parallel execution. Once the correct ordering of these iterators is determined they are chained together into trees where the output of one iterator is the input to another iterator. For a reference to the steps in which a query takes during standard processing refer to Figure 2.2. As a concrete example of this process take the example query (taken from [12]):

```
SELECT S.name
FROM Reserves R, Sailors S
WHERE R.sid = S.sid
      AND R.bid = 100 AND S.rating > 5
```

Which can be expressed in the relational algebraic form:

$$\pi_{sname}(\sigma_{bid} = 100 \wedge rating > 5(Reserves \bowtie_{sid=sid} Sailors))$$

And finally translated to a query plan of iterators shown in Figure 2.1.

2.3 Current research

It is known that memory access is a main source of bottlenecking in allowing modern CPUs to operate at their full capacity. In order for the memory accesses to be modified a drastic change would need to made in the layout of databases. In recent years researchers have turned their attention to other ideas, such as compiling queries to machine specific executables, to increase the runtime performance of query execution. IBM researchers presented both a compiled and interpreted query execution

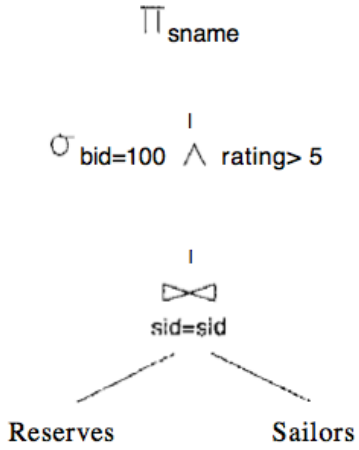

Figure 2.1: Query evaluation plan. Taken from [12].

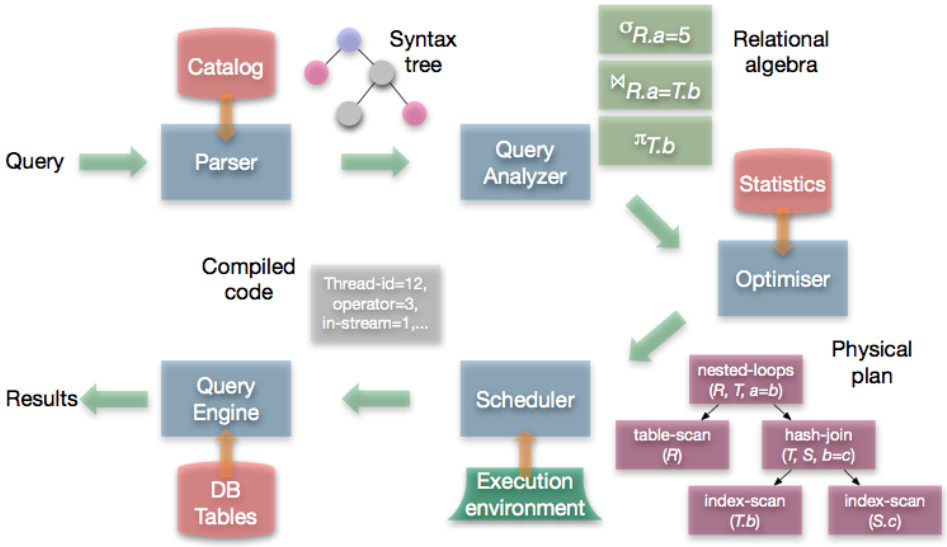

Figure 2.2: Query cycle. Taken from http://www.inf.ed.ac.uk/teaching/courses/adbs/slides/adbs.pdf

model, using the Java Virtual Machine (JVM), in [13] which demonstrated significant improvements over the standard SQL Virtual Machine (SVM) currently in use. This paved the way for other researchers to begin looking at compiling database queries.

Another approach that was taken in this area involves using code templates for various query algorithms present in database queries [9]. The idea is that by using code templates for certain query algorithms much of the bloat associated with the current database models can be minimized. In addition to this, code can be generated and optimized for the specific machine on which it will be running. While this approach does not completely separate itself from the operator model of query execution it does make a drastic change from the iterator model. Results of this study show a very large improvement in runtime with the trade off being that the query plans take longer to emit because of the increased time required for compilation [9]. This research follows a classical query execution plan until the point directly following the query optimizer (Figure 2.3). At this point the system executes a phase of code generation. During this phase the system receives a list of operator descriptors that are determined by the query optimizer. These descriptors include, among other details, algorithm information for each operator. Next the system will look at each of these operator descriptors and will retrieve the corresponding C code template which is instantiated with that operators descriptor. Once all the operators have been processed in this fashion a single source file, which contains all the functions generated from each operator, is created. This source file can then be compiled for the specific machine on which it is running and executed to produce the result of the query.

A similar strategy was presented in [11] in which queries are first optimized and then compiled for the target machine. One key difference in this proposal is that the operator model is less present in the execution plan. This project also does not use a template plan for code generation but translates the queries to LLVM assembly with some C++ code used for complicated portions. Evaluation of this system followed a

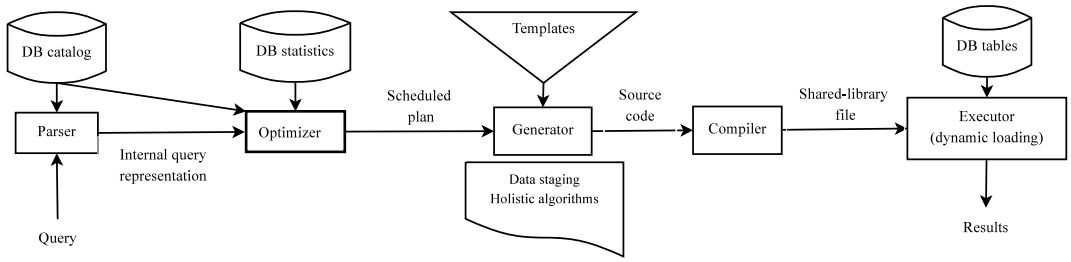

Figure 2.3: Query execution plan from [9].

strategy of measuring the proposed execution model's runtime on several benchmarks and comparing this time with the same benchmarks run with various database systems. A comparison was also made between the compile time of the generated LLVM assembly code and previously hand coded C++ queries. This showed a very fast compilation time and runtime for code produced in LLVM.

One of the major changes in this project is the idea of a *pipeline breaker*. A main advantage of the iterator model is that it allows for very straightforward pipelining between operators. However, when the iterator model is not the model being used pipelining becomes much more of a challenge. Here the researchers have identified the concept of a *pipeline breaker* as any point in the execution plan where data must be spilled to memory. This concept can therefore extend the pipeline between operators. The model designed from this therefore proposed to keep data in the CPU registers for as long as possible by pushing tuples from one *pipeline breaker* to another. Once the query is broken up into these segments pseudo-code can be generated for the different sections which is then translated into mostly LLVM assembly (with pre-compiled C++ code being used for complex operations such as handling page I/O).

A comparison of various state of the art techniques can be found in [15]. In this report a detailed analysis of three expressions is presented: projection, selection, and hash join evaluated using various compilation, vectorization, and a mix of both is presented. The findings from this investigation point in the direction that neither query compilation or vectorization is singularly best for optimal query execution. Rather a combination of the two produces better results.

Chapter 3

Design

The overall design of goas is more akin to a compiler driver than that of a straightforward compiler. When looking at a standard compiler the chain of actions is the compiler accepts a piece of source code and, more or less, directly emits a machine specific executable. Whereas a compiler driver will have several stages in between the reading of source code and the emitting of an executable file. The GNU C Compiler (gcc) is a classic example of this. When a C source file is passed to gcc there are three main stages of compilation. First, the source file is translated to an assembly file which is then turned into an object file and finally linked into an executable file [8]. The goas compiler also follows a non-traditional compilation plan.

3.1 Optimized Algebraic SQL (oas)

This language was designed with the assumption that the source code would be produced by the query optimizer in a database management system (DBMS). In addition to this only unary operators were considered for this implementation (i.e. scan, select, project, and sort). The syntax for the language is defined as follows:

```
tokens:
NUM     := [0-9]+
KEY     := [a-zA-Z]+\.[a-zA-Z0-9]+
INPUT   := [a-zA-Z][0-9a-zA-Z]*
ATT     := [a-zA-Z][0-9a-zA-Z ]*
FP      := [\_a-zA-Z0-9\/.-]+
PRED    := [&  |]
OP      := [= < >]
```

```
rules:
program := line
line    :=   ε
           | line INPUT ':=' scan '\n'
           | line INPUT ':=' select '\n'
           | line INPUT ':=' project '\n'
           | line INPUT ':=' sort '\n'

scan    := 'scan(' FP ');'
select  := 'select(' INPUT ',[' selectOp ']);'
project := 'project(' INPUT ',[' keylist ']);'
keylist :=   ε
           | KEY
           | KEY ',' keylist
           | INPUT
           | INPUT ',' keylist

selectOp :=   ε
           | PRED ',' selectOp
           | '(' KEY ',' OP ',' KEY ')'
           | '(' KEY ',' OP ',' KEY ')' ',' selectOp
           | '(' KEY ',' OP ',' NUM ')'
           | '(' KEY ',' OP ',' NUM ')' ',' selectOp
           | '(' KEY ',' OP ',' ATT ')'
           | '(' KEY ',' OP ',' ATT ')' ',' selectOp
           | '(' INPUT ',' OP ',' INPUT ')'
           | '(' INPUT ',' OP ',' INPUT ')' ',' selectOp
           | '(' INPUT ',' OP ',' NUM ')'
           | '(' INPUT ',' OP ',' NUM ')' ',' selectOp
           | '(' INPUT ',' OP ',' ATT ')'
           | '(' INPUT ',' OP ',' ATT ')' ',' selectOp
```

A sample program which implements the query:

```
SELECT rating, name
FROM sailors
WHERE rating>5 OR name='long john silver'
ORDER BY rating
```

Can be seen in Listing 3.1.

Listing 3.1: Sample query for the OAS language.

```
x0:=scan(src/tables/sailors.txt);
x1:=select(x0,[(rating,>,5),|,(name,=,long john silver)];
x2:=project(x1,[rating,name]);
x3:=sort(x2,[rating],ASC);
```

3.2 Compilation Plan

There were two paths of compilation considered during the project. The first was to build a compiler front-end for gcc. A front-end in this case, for gcc version 4, is a parser that produces an intermediate representation (IR) that gcc knows, called RTL. From here gcc will apply various optimizations and produce a machine specific executable [8]. In order to do this a front-end will parse a language source file making calls to a GNU Compiler Collection (GCC) language knows as GENERIC. These calls should perform the various actions that each part of the language requires. After this the code that is emitted from these calls will be translated into another language inside of GCC called GIMPLE which is then finally translated into the RTL that gcc needs to produce an executable. While this is a viable option for producing a compiler there are many negative aspects in relation to this project. For starters, the files required to interface with GCC and making the interface fully functional is fairly complicated and not extensively documented. Though possibly manageable in the time frame of this project it is likely that other aspects would have suffered due to time spent working on interfacing with GCC. In addition to this the language required to perform this, GENERIC, is not a very well known language. This would have slowed down production for the time to learn the language. In addition to this any future work on the project would need to be done in this language which is not very appealing. For these reasons this path was not chosen. Instead the goas compiler takes a much more straightforward approach and compiles everything to the C programming language.

Compiling a query happens in two stages. The first stage is the query is passed to the parser which emits a C program. This phase of compilation will only catch syntactic errors in the program and does nothing in the way of error reporting other than to alert the user if there is a syntax error. This program is then compiled and linked to a static library which contains all the code for the various operators. At this point the .c file can be compiled with any C compiler. For this project all files were compiled and linked to the oas library with gcc using the command 'gcc temp.c -Llib/ -Iinclude/ -loas -o temp'. With temp.c being the .c file created by the parser. Figure 3.1 illustrates the various stages that a query would take beginning with the query optimizer.

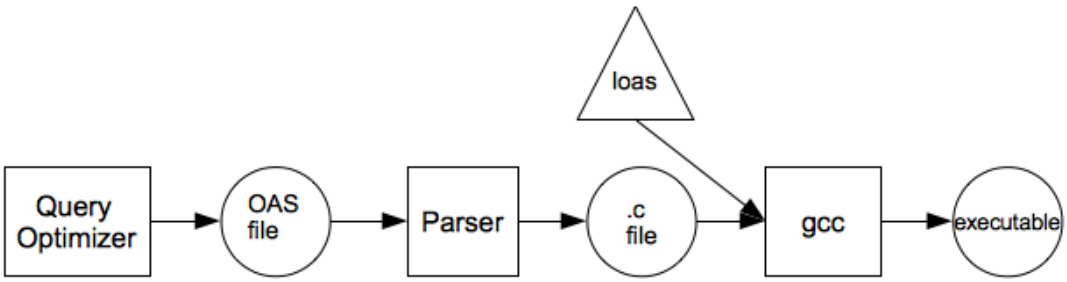

Figure 3.1: Compilation plan for the goas compiler.

Chapter 4

Implementation: GOAS

Goas is comprised of two main parts: the parser and the library of operators. The parser and library are pre-compiled so these don't need to be changed during execution. Parsing is accomplished with two files: oas_scanner.l and oas_parser.y which are compiled and linked to create an executable. To parse a file simply 'cat' the file and pipe that output to the executable. This will generate the corresponding .c file. The library is created by linking object files for each of the operators into one library (record.o, scan.o, select.o, project.o, sort.o). In order to use this library the appropriate headers need to be included and it can then be linked by whatever library linking convention the compiler being used requires. Also, for testing purposes it was necessary to include two other source and header files: getdelim and getline. These are generally included in the standard runtime library but were not available on the operating system used for testing purposes. These were packaged into the library.

When a query first begins there is no known information about any of the relations that are being used. However, once scanning of a table is finished some information about the table is gathered and maintained globally so that all operators can use this information. Namely, the number of records and the number of columns currently in the relation. This information is kept up to date as each operator modifies the relation.

4.1 Scanning and Parsing

Scanning and parsing were written using flex and bison, respectively. The scanner is very straightforward and simply returns the tokens specified in the language (e.g. Input, key, num). The parser for goas, rather than emitting some kind of tree shaped intermediate representation, generates a temporary C file (temp.c). As a query file is

being processed the parser will emit lines of C code that correspond to the appropriate function calls required to process the query. Looking back to Listing 3.1 this query would be parsed into the following C file:

Listing 4.1: C code emitted for sample query.

```
#include <stdio.h>

#include ``record.h''
#include ``scan.h''
#include ``select.h''
#include ``project.h''
#include ``sort.h''

int main() {
  r_list *x0 = scan(``src/tables/sailors.txt'');
  const char *keys0[8] = {``rating'', ``>'', ``5'', ``|'',
    ``name'',``='', ``long john silver'', NULL};
  r_list *x1 = select_oas(x0,keys0);
  const char *keys1[3] = {``rating'', ``name'', NULL};
  r_list *x2 = project(x1,keys1);
  s_order order0 = ASC;
  const char *keys2[2] = {``rating'', NULL};
  r_list *x3 = sort(x2,keys2,order0);
  print_r_list(x3);
  return 0;
}
```

This is accomplished by walking through the source file provided and translating the various portions to the appropriate C code. The first action the parser will take is to include the header files associated with all the operators and set up a main function. The parser will include the header files for operators even if they aren't present in the query. Now the parser can look at the source file. Each new line in the source file is going to be an assignment to a new relation, so the parser will first create a pointer to a new r_list variables when it first encounters a line. At this point the parser will see what algebraic function is to be executed next. In all cases except for scanning, the parser will need to handle the list of parameters required for that function. Once the parser has

created the necessary parameters and written everything that the function will need to the file, it will then emit the code to execute that function. This approach is somewhat naive in the fact that a new pointer is created for each relation and a new list of keys is generated for every function, regardless of if that list has been used before. For simple queries this isn't likely to cause any performance degradation but if there were queries with very large lists of keys that were used repeatedly, or the same relation being used in lots of operations, then memory would be allocated for the same purpose multiple times.

This assumption is made mainly because the parser has no concept of schemas or that it is dealing with database queries at all. The parser's only goal is to translate the file presented to it in oas to a C program which will execute the query described in that file.

4.2 Operators

Relations are represented as two C structs: a record and a relation(r_list). Records are made up of an array of strings that contain the data for each attribute in that record. This array is dynamically allocated during the scanning phase of a query. In addition it will be dynamically reallocated during the projection phase (assuming the projection removes some columns). An r_list is made up of two arrays: the names of each column and an array of the records in this relation. These arrays are also dynamically allocated during the scanning phase (and reallocated during selections and projections as needed). The reasoning behind this is that at this stage of the query execution it's unlikely we will have accurate information about the size of the table and how much memory will need to be allocated for each record. However, it is likely the query optimizer will be able to provide an estimate. This estimate is represented in the initial memory allocations for the arrays. If, for whatever reason, these estimates are wrong the arrays can be reallocated to fit the actual data. Each of the operators present in the language are defined by:

scan Takes as its parameter the name of a file. It reads this file, creates the appropriate C structures, and allocates the memory required to manipulate a relation. Finally, it returns a pointer to the relation.

select The components that make up the parameters for a selection are:

Input This is the name of the relation to be used in the selection.

key This is the name of a field in the specified relation.

op The operator to be used for comparison. Currently the only supported operators are $=$, $>$, and $<$.

str/num Either a string or a number is required here.

connector Is a logical operator. Currently only & and | are supported.

The number of these parameters isn't limited. Select performs the specified comparison in left to right order on the relation and returns a pointer to the relation with the records that didn't match the selection criteria removed.

project Takes as parameters the name of a relation to be used for the projection and a list of keys (i.e. field names). Each record in the relation will be modified and only the keys specified will be kept. Finally, a pointer to the modified relation is returned.

sort Takes as parameters the name of a relation to be sorted, a list of keys on which the relation will be sorted, and a specification as to whether it should be sorted in ascending or descending order. Finally, a pointer to the modified relation is returned.

During the scanning process there is a significant amount of memory allocations and various other data setups that are taking place. The majority of this in handled by the file record.c which consists of the following functions:

r_list *init_r_list(char **) This function takes as its parameters a list of strings that make up the names of each of the columns in the relation. It will allocate the memory needed for an r_list, the memory needed for an array of records, and the memory needed to store each column name (as well as copy each column name into the array). In the end it returns a pointer to the initialized r_list.

void create_records(r_list *, char *, FILE *) Create_records() takes as its parameters a pointer to the relation, a pointer to the first line (after the names of the columns) of the file pointed to by the third parameter. This function goes line by line through the file separating each line into the appropriate fields, allocating memory for that entry, adding it to the current record, and finally adding that record to the end of the records array of the relation passed as the first parameter.

void add_record(r_list *, record *) This function takes as its parameters a pointer to the r_list to which this record should be added and a pointer to the record. It first makes the checks necessary to ensure adding this record will not exceed the bounds previously allocated for records. If adding the record will be in bounds it is added to the records array and the current record count is increased. If not, more memory is allocated for records and the record is added.

void print_r_list(r_list *) This function takes as its parameter a pointer to the r_list which is to be printed. It then prints this in a format similar to that of PostgreSQL freeing memory along the way.

void check_malloc(void *, char *) Takes as parameters a void pointer and a string which will be printed if there is an error. This function is nothing more than a convenient way to make sure a memory allocation was successful and exit the program if it failed.

4.3 Variations

In addition to the implementation described in the previous section two slightly alternate versions of goas were also produced. In the first OpenMP is used in a very rudimentary and experimental form to parallelize chosen operators. The second variation only changes how files are scanned into the system. In this implementation a memory map is used for file manipulation in the initial scanning process.

4.3.1 OpenMP Parallelization

There were two main choices of parallel libraries that were considered for this project: pthreads and OpenMP. The choice to use OpenMP was based mainly on its comparative simplicity. In order to parallelize code with OpenMP all the programmer needs to do is indicate sections that can be parallelized. Then it is left up to the runtime system to determine the most appropriate way to do this. With this freedom a query engine that was compiling queries could be programmed to spot areas of parallelism and insert OpenMP tags into the generated code. OpenMP is also a widely used framework and available on various different platforms which increases the portability of the code.

OpenMP is an API for shared memory parallel programming supported across multiple platforms [3]. This API provides the user with a variety of directives that can be

inserted into portions of their code that provide the compiler with the necessary information to perform these blocks in parallel. In addition to this OpenMP provides runtime library routines and environmental variables that can be used in the program [2]. In order for a portion of code to be executed in parallel it must be enclosed inside a *parallel region*. This parallel region allows the user to specify features such as variables that are private to each thread, variables that are shared between all threads, the number of threads that will execute in the region, as well as many other features. When a thread reaches the beginning of a parallel construct it will fork the number of threads specified (or the number determined by the runtime system if there is no specification) and the threads will join again at the end of the region [2]. In addition to this the OpenMP API provides *work-sharing constructs* that allow for different types of parallelizations. The only work-sharing construct that was used in this project is the *DO / for directive*. This construct allows for iterations of a specified loop to be executed in parallel and enables the user to specify various parallelization features in terms of scheduling, the order of iterations, private variables, shared variables, and other features. The only other OpenMP directive used was the *critical* construct which is used to specify a section of code in which only one thread is permitted to be executing at any time.

The only two operators parallelized in this project were select and project. These operators exhibit very clear data parallelism in that when performing a selection or projection each record can be evaluated or modified independently of the others. This is the approach taken to parallelizing these two operators. Each thread is repeatedly given one record to manage until there are no more records in the relation. For all the parallelisms created all possible user supplied specifications are left to the runtime system except the scheduling which is specified to by dynamic. This means that each thread is given an iteration of whichever loop is being executed and when they are finished they are dynamically assigned another iteration [2].

4.3.2 Memory Mapped Scanning

Memory mapping is a technique used to reserve pages of memory for a specific object. In order to do this the user must specify the address corresponding to where the mapping will start, the length of the mapping, flags for the runtime system to determine the starting address, protections upon which the object will be subjected, the object, and the offset into the object where the mapping will begin [5].

For this project, memory mapping was used during the initial scanning of a file. A memory map is created with the object being the file that contains the table, from which memory is read byte by byte. This differs from the previous version in that previously files were read line by line instead of byte by byte. In this version all the parsing is done manually with memory being allocated in the same fashion.

Chapter 5

Evaluation

All tests were performed on a MacBook Pro with a 2.2 GHz Intel Core 2 Due Processor, 6GB of memory, running Mac OS X 10.6.8 under the user ">console". Testing for the goas compiler was accomplished in the following stages:

1. Purge the disk cache
2. Compile the query (obtain the time required to compile)
3. Execute the query (obtain the time required to run) and route the output to /dev/null to minimize the print time.
4. Repeat 10 times (gather average and standard deviation)

Step 1 in this plan is used to simulate a cold run in the system as purging the disk cache should approximate a fresh boot [4]. To compare the execution times all the queries were also run on the same hardware running a PostgreSQL server. These tests followed the same outline as previously presented with the only difference being there was no step 2 for these queries. In addition to this the suite of queries was also executed with the goas library compiled under the -O2 gcc optimization flag. The queries were also executed for the parallel variation and memory mapped variation both of which were compiled under the -O2 optimization flag.

The PostgreSQL server used for testing was 64-bit PostgreSQL 9.1.3 initialized with all the default settings. These include an 8MB shared memory buffer, 100 maximum concurrent connections, maximum execution stack depth of 2MB, and 1MB of memory to be used for internal sort operations and hash tables.

All tables used for evaluation were generated using a modified version of the code that can be found at [1]. This is a table generator for relations that largely conform to

Column name	Type
unique1	long
unique2	long
two	long
four	long
ten	long
twenty	long
onepercent	long
tenpercent	long
twentypercent	long
fiftypercent	long
unique3	long
even	long
odd	long
stringu1	string
stringu2	string
stringu4	string

Table 5.1: Schema description for tables used in query evaluation.

the Wisconsin Benchmark, for which information may be found at [6]. The relations generated from this follow the form show in Table 5.1.

5.1 Queries

For both systems queries were run that analyzed each operator separately with increasing selectivity and increasing table sizes. All queries were run on tables ranging from one record to one million records (in gaps of 100,000). An example of the selection queries for a table of 100,000 records (t100k) can be seen in Listing 5.1.

Listing 5.1: Selection queries for PostgreSQL.

```
psql -d postgres -c ''select * from t100k where two=1;''

psql -d postgres -c ''select * from t100k where twenty<9
OR twenty=18;''
```

```
psql -d postgres -c ``select * from t100k where twenty<8
OR twenty=9 OR twenty=17;''
```

```
psql -d postgres -c ``select * from t100k where
onepercent<48 OR onepercent=90 OR onepercent=20 OR
onepercent=50;''
```

```
psql -d postgres -c ``select * from t100k where ten=1
OR ten=2 OR ten=3 OR ten=4 OR ten=5;''
```

To balance the time that would be spent printing all selection queries were designed to select half of the records in the table. A similar approach is taken to projection and sorting queries, with five queries of increasing complexity being executed. Finally, five more queries that follow a plan of scan, select, project, sort were tested. These queries are made up of the individual select, project, and sort queries previously tested.

Charts for all tested queries are provided in the following sections. They show the selectivity on the x-axis, average runtime across the on the y-axis, and have the standard deviation as y-axis error bars.

5.2 Analysis

Overall analysis of the various versions tested appears to indicate that for scanning, selecting, and sorting the memory mapped version of goas performs the best. However, with projections and more complicated queries PostgreSQL appears to be superior. In general it appears that all versions of the compiled queries are relatively unaffected by the amount of predicates or keys used in any specific operator. In addition to this it appears the queries executed on the PostgreSQL DBMS suffered from a larger error range than those of the compiled queries.

In general these results conform with what would be expected from such tests. In looking only at the compiled versions the non-optimized version doesn't perform as well as the version that does have the optimizations. Furthermore, the version that includes compiler optimizations but handles file manipulation in a more advanced way outperforms the version with just optimizations. As this is what optimizations are meant to do these results aren't at all surprising. For the version the used OpenMP

there were really no expectation as to performance other than it would either succeed and provide some decent improvement or the parallelizations wouldn't be appropriate and would cause the execution to be slower. In this case the parallelizations didn't work out and thus caused the code to slow down. As an overall comparison between PostgreSQL and compiled queries it is not surprising the compiled queries often run quicker. The argument earlier was that when a query is executed on a standard query engine there is an excessive amount of function calls that need to happen to get information between operators. These function calls cause the processor to perform sub-optimally. In the compiled versions of all the queries there is no large amount of function calls required to access the records. They are all stored in arrays in main memory and can be access simply by reading from, or writing to, the correct memory location. This directly illustrates the benefits that can be obtained by breaking free of the iterator model described earlier. Looking at each query that used a table size of one, the bloat of PostgreSQL can very clearly be seen. For each of these examples the time difference between all the compiled versions and PostgreSQL is significantly larger than any of the other queries. Even though the table only consists of one record PostgreSQL is still going through all of the stages of the query engine. But none of these stages are really necessary. With a compiled query it is simple and bloat-free to read a small file and apply various transformations.

To analyze the time for compilation of queries the average was taken across all table sizes for each query. This is because the table size really has no impact on the compilation time. Looking at Figure 5.1 it can be seen that the average time to compile ranges from approximately 2.2 seconds to 2.5 seconds. In addition to this the chart shows that the time to compile is mostly independent of the query. This is because the source code that needs to be compiled to execute a query is very compact and contains between one and three lines to execute an operator. It can also be seen that the memory mapped version takes the longest to compile. This is likely explained by the extra libraries that need to be included in order to manipulate a file as a memory map. The main thing to note about the compilation time of a query is that this is a one time cost as this executable will be stored and used again when that query is needed. In this way the time to compile becomes much less of a factor.

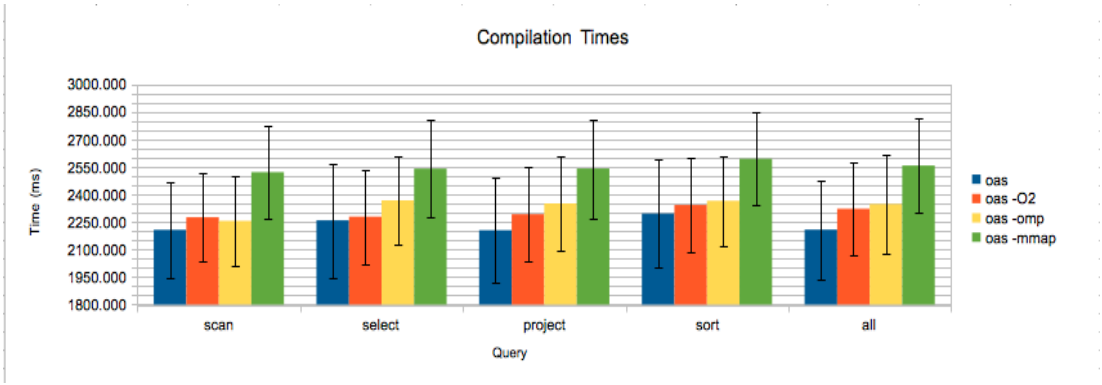

Figure 5.1: Times for compilation for tested queries.

5.2.1 Scanning

Looking at Figure 5.2 shows that scanning was most effective under the memory mapped version of goas with the difference in times between the PostgreSQL evaluation and goas versions diverging more and more as the table size increased. This is somewhat surprising since it is the case in most of the other queries tested the performance of PostgreSQL is better with larger files. Figure 5.3 shows the standard deviation across the different table sizes. There are two large spikes at the 1,000,000 record table for PostgreSQL and the goas version compiled with no optimizations. It can also be seen that the variation in scan times for the memory mapped version is very low and much less sporadic than the other versions. The explanation for this likely lies in the fact that once the file is memory mapped there is much less work that needs to be done in order to access the file. When a file is memory mapped the only overhead in accessing a file is if the operating system has to bring the page which is to be read from into main memory. However, when using standard I/O there are several computations that need to be made in order to get the right location in the file. These computations and checks are likely to take different amounts of time during each run which was surely a contribution to the difference in standard deviation.

These results of course make sense and directly support the arguments made previously for compiling queries. In order for this query to execute on PostgreSQL it must go through the entire query engine process. This means the query will be have to be parsed, analyzed, optimized, scheduled, and finally executed. This is certainly going to take some time. However, with the compiled version the process is very simple: read this file from the disk and allocate memory for its contents. The amount of computation that needs to be done for each system is very different and can be seen in the

result times.

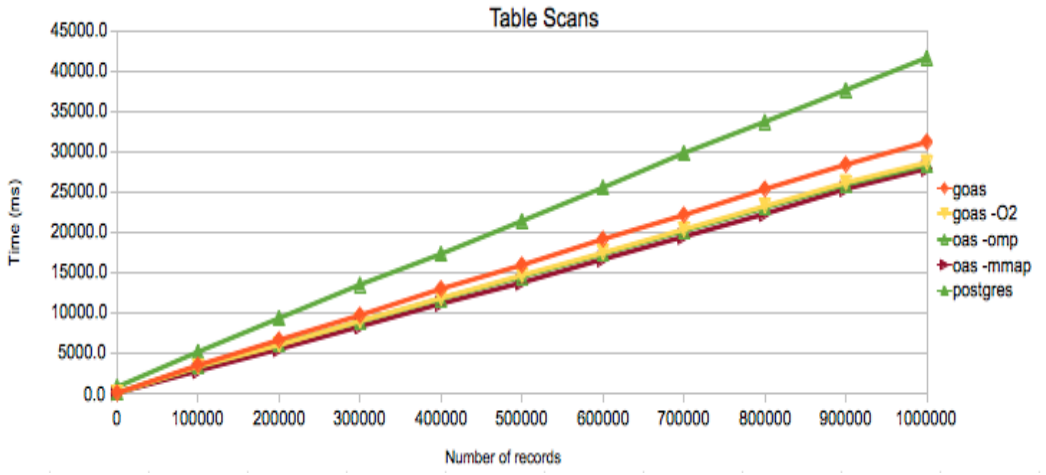

Figure 5.2: Scanning time for tested table sizes.

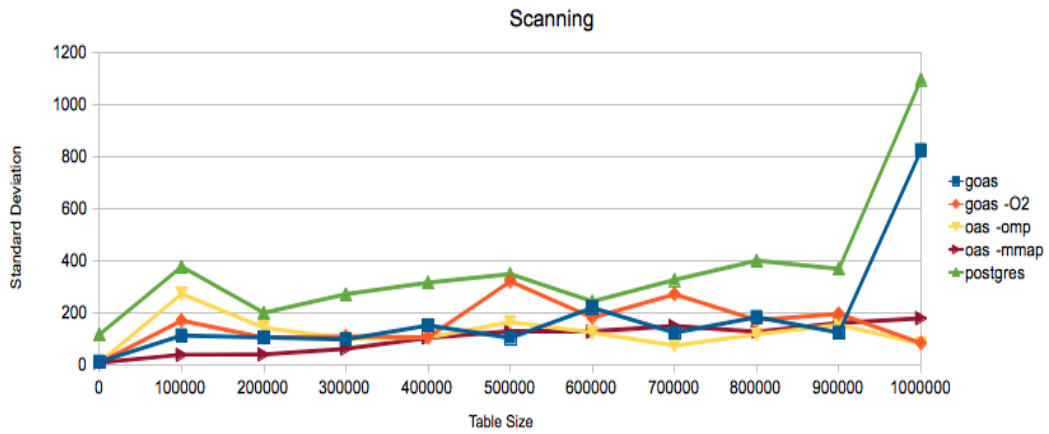

Figure 5.3: Standard deviation for scanning.

5.2.2 Selection

All charts for these queries can be found at the end of this section. These charts show that for all queries of all table sizes the memory mapped version of the goas compiler outperforms all other test systems (except for table size of 200,000 and selectivity of 1 where the non-optimized version is the fastest. This is likely a fluke in the data). For smaller table sizes the difference between the compiled queries and PostgreSQL is much larger. Figure 5.4 shows the select queries performed on a table with only

1 entry. In this case PostgreSQL is at best 17 times slower than any of the compiled queries (selection with 4 predicates). As the table sizes increase PostgreSQL begins to outperform the regular version, -O2 version, and the version with OpenMP but doesn't outperform the memory mapped version in any query on any table size. It is also clear from looking at the selection charts that the estimation in time for the memory mapped queries is very accurate. Where in many of the charts the standard deviation error bars have a somewhat large range the error bars for the memory mapped version are consistently very small.

The performance of selection in the compiled queries can be explained by similar reasons to that of scanning. In addition to the overhead necessary in getting the query through the engine PostgreSQL is also suffering from the excessive amount of function calls explained previously. For every tuple that needs to be processed by the select operator a function call needs to be made to the scan operator to obtain the tuple to be processed. The scan operator will then give the select operator the next tuple and this will repeat until the relation is out of tuples. On the other hand, in the compiled version the relation is loaded directly into main memory and the select operator accesses each tuple via array indexes. In addition to this since the change from one tuple to the next doesn't result in a function call, and thus clear the registers of the processor, a smart compiler will be able to keep various values in the registers as it loads records from memory. This type of access is much faster than the repeated function calls used in a standard query engine and is shown in the select results provided.

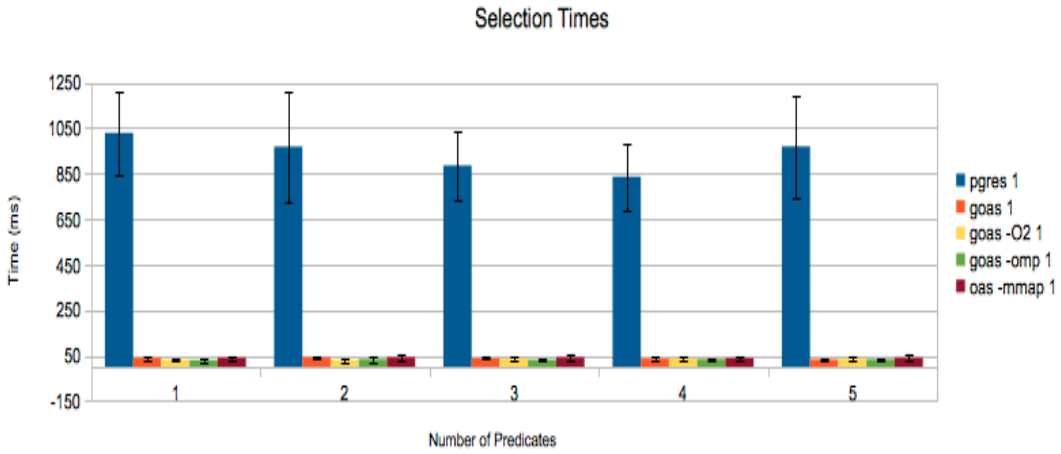

Figure 5.4: Selection times for table size of 1.

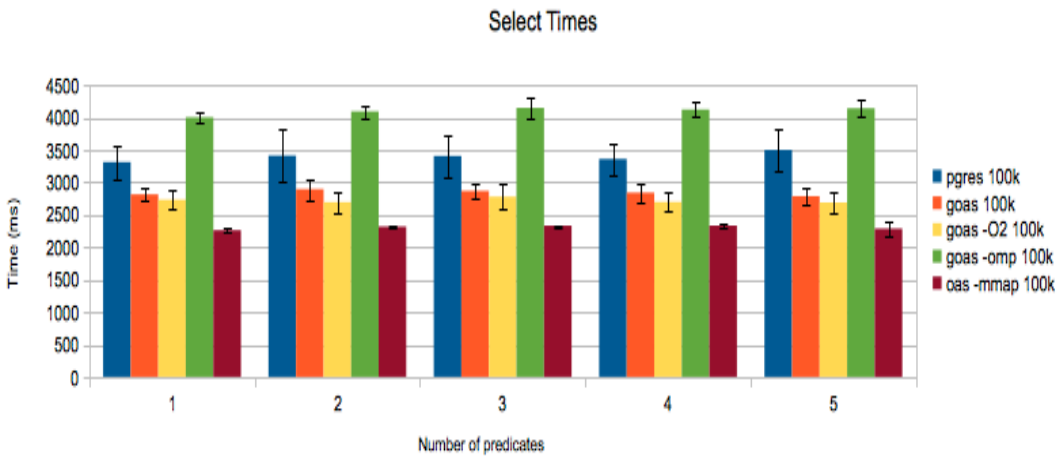

Figure 5.5: Selection times for table size of 100k.

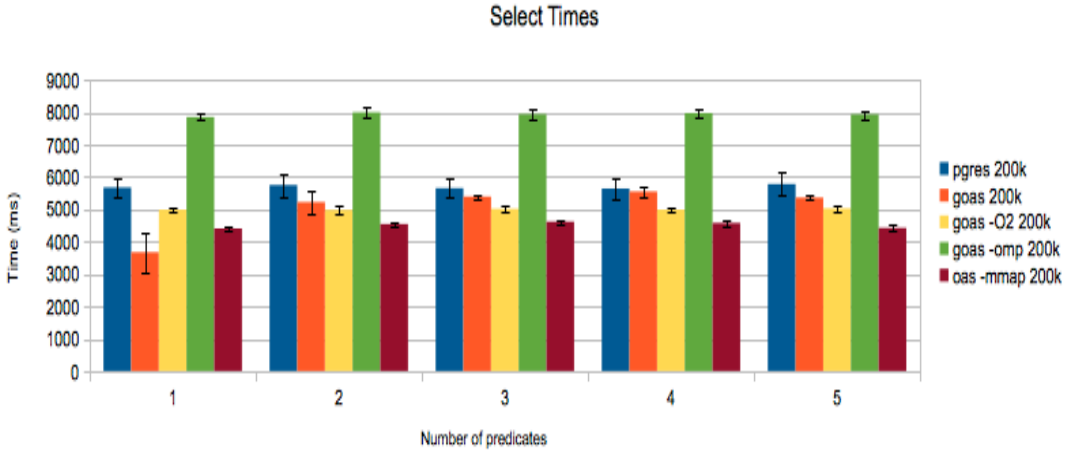

Figure 5.6: Selection times for table size of 200k.

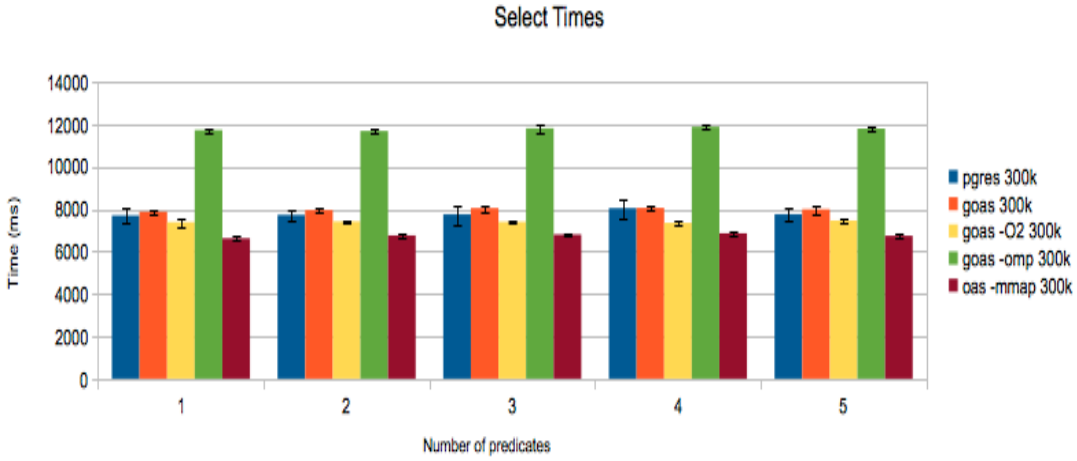

Figure 5.7: Selection times for table size of 300k.

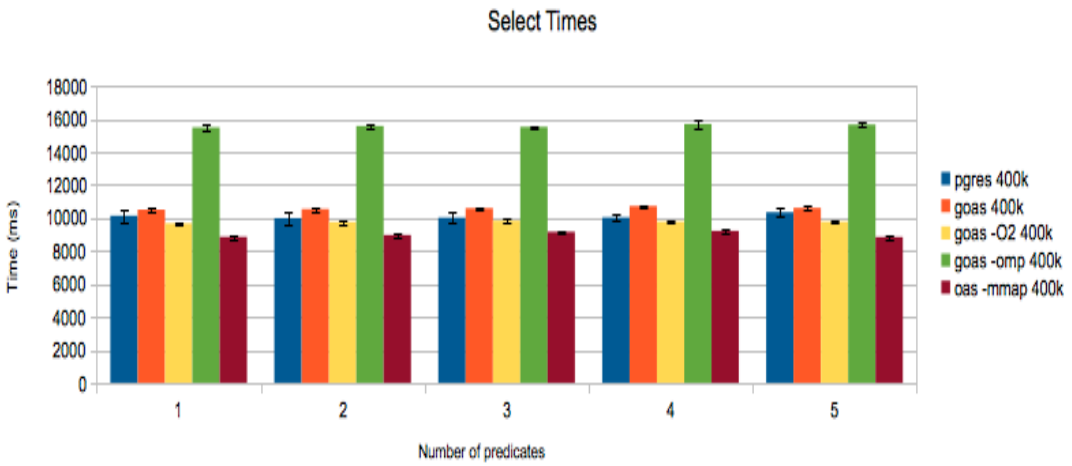

Figure 5.8: Selection times for table size of 400k.

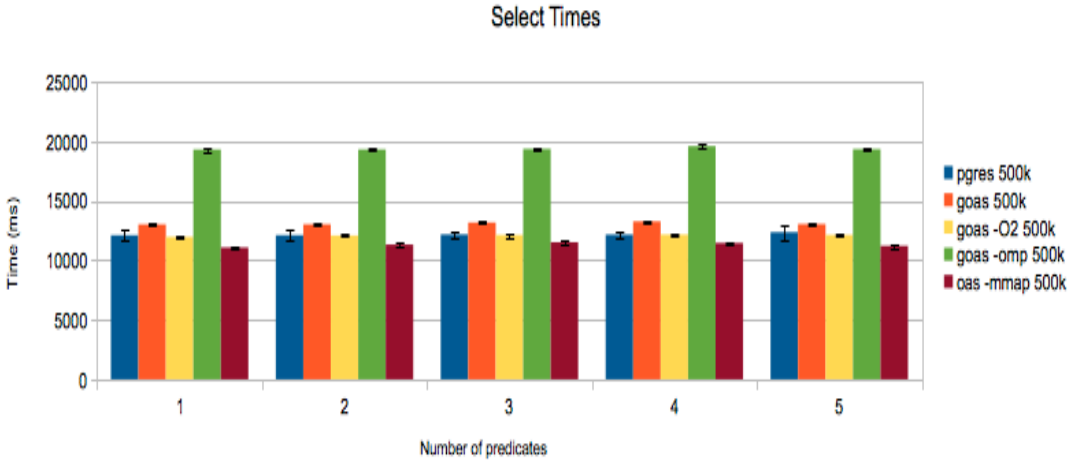

Figure 5.9: Selection times for table size of 500k.

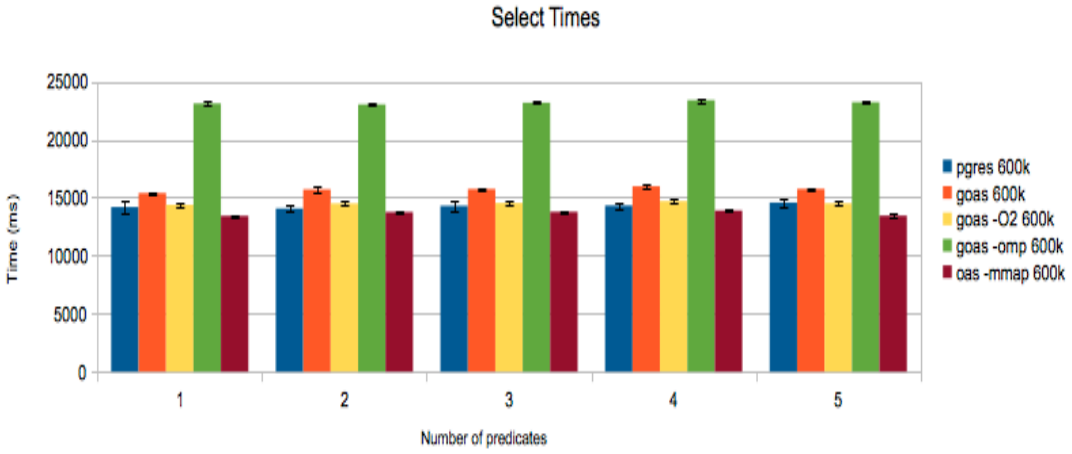

Figure 5.10: Selection times for table size of 600k.

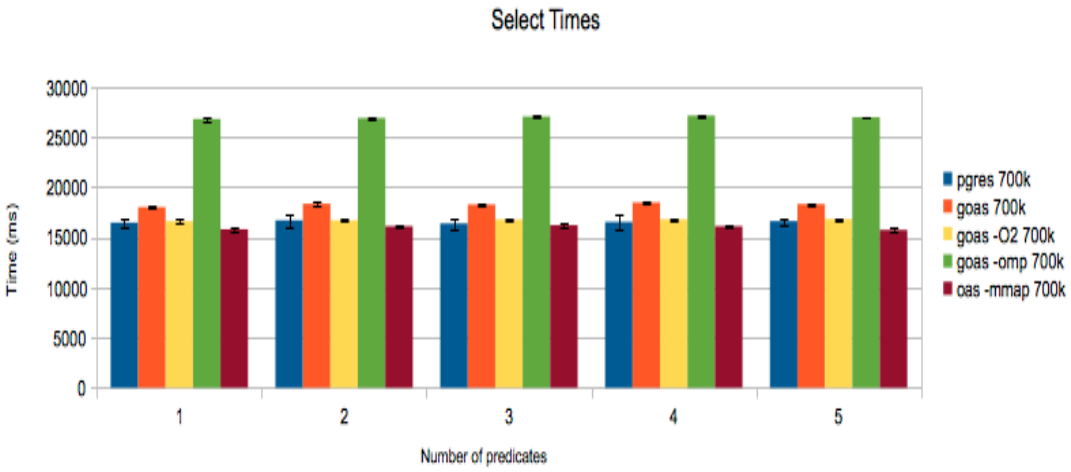

Figure 5.11: Selection times for table size of 700k.

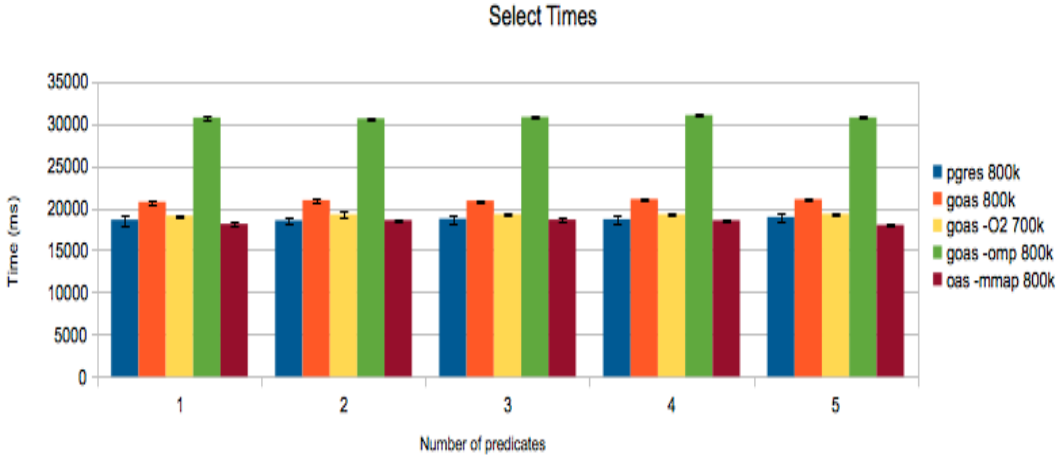

Figure 5.12: Selection times for table size of 800k.

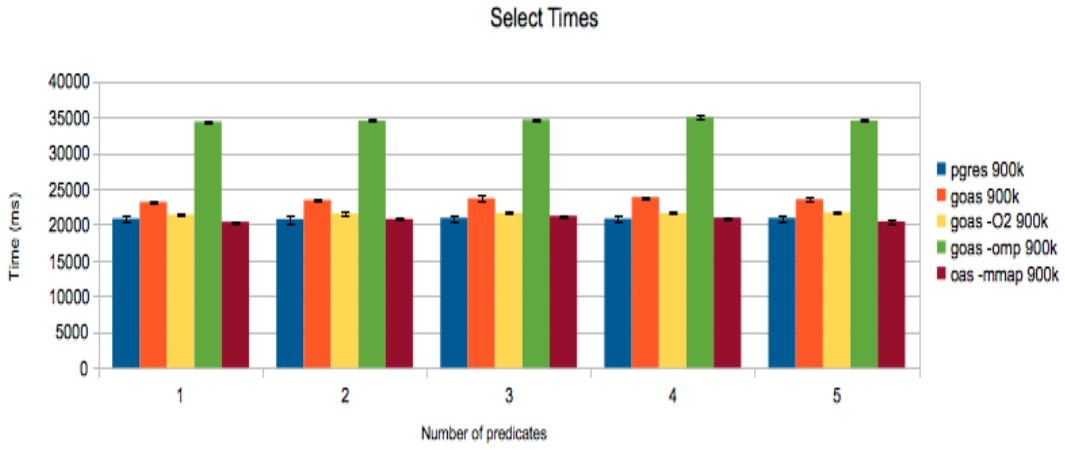

Figure 5.13: Selection times for table size of 900k.

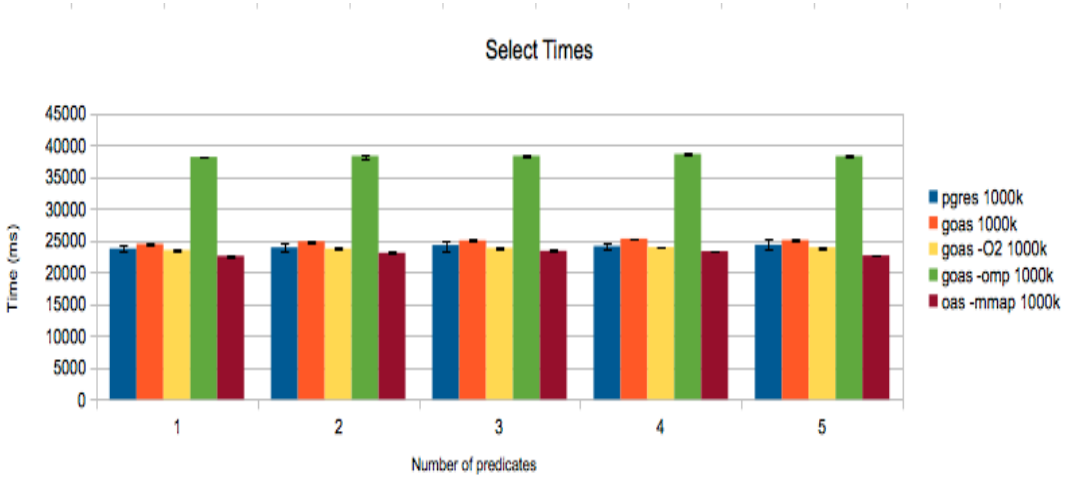

Figure 5.14: Selection times for table size of 1000k.

5.2.3 Projection

All charts for these queries can be found at the end of this section. Looking at these charts it can be seen that projection evaluation in PostgreSQL is heavily affected by the number of keys in a projection. In most cases it takes almost three times longer for projections involving twelve keys as opposed to one key. While PostgreSQL does outperform the goas compiler on queries involving table sizes over 100,000 it never outperforms on queries involving twelve keys and is only slightly faster on queries with nine keys. As was the case with selection it can be seen here that on queries with one record PostgreSQL has performance no where near the compiled queries. Also as in the selections the error range of the memory mapped goas queries is consistently much smaller than any of the other systems.

For compiled projections the results are similar to those of selections. Due to the fact that all tables are stored in main memory there is really no difference between projecting on one key or projecting on twelve keys. This can be seen in the charts as there isn't a great deal of difference between one key and twelve (not including the OpenMP version). The results of projection in PostgreSQL are somewhat a mystery. It is unclear what happens in PostgreSQL after a table size of 100,000 that makes queries with less keys faster than the compiled queries. There is also a very unusual pattern in the time a query takes as the number of keys is increased.

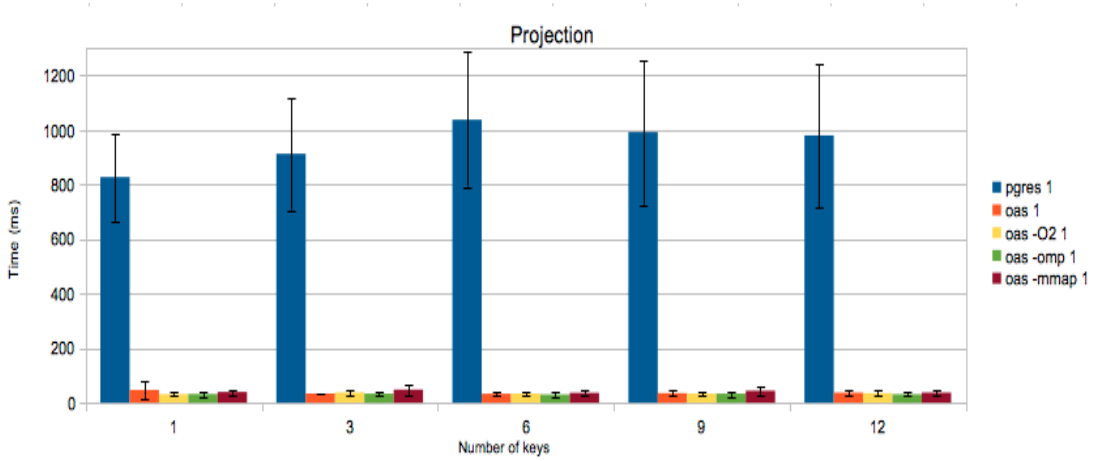

Figure 5.15: Projection times for table size of 1.

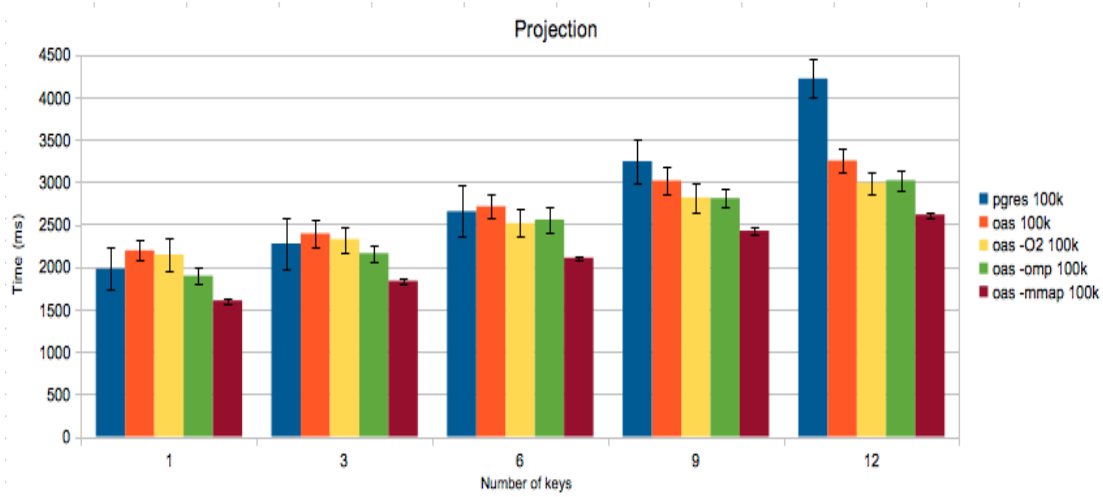

Figure 5.16: Projection times for table size of 100k.

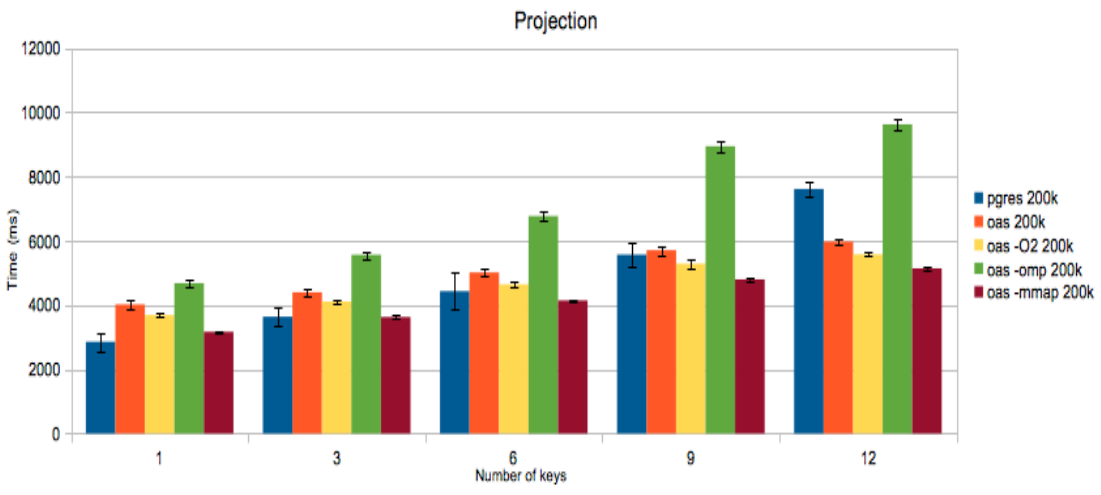

Figure 5.17: Projection times for table size of 200k.

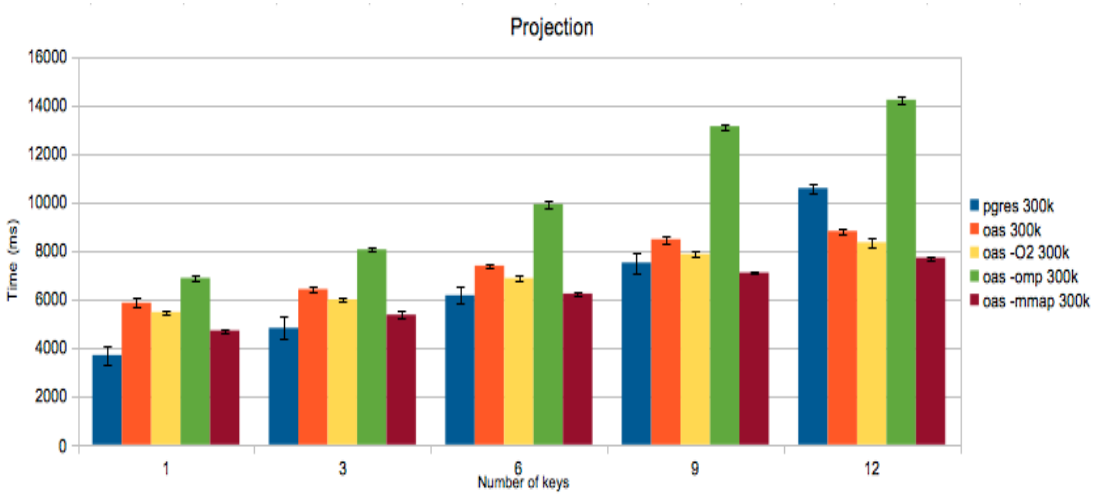

Figure 5.18: Projection times for table size of 300k.

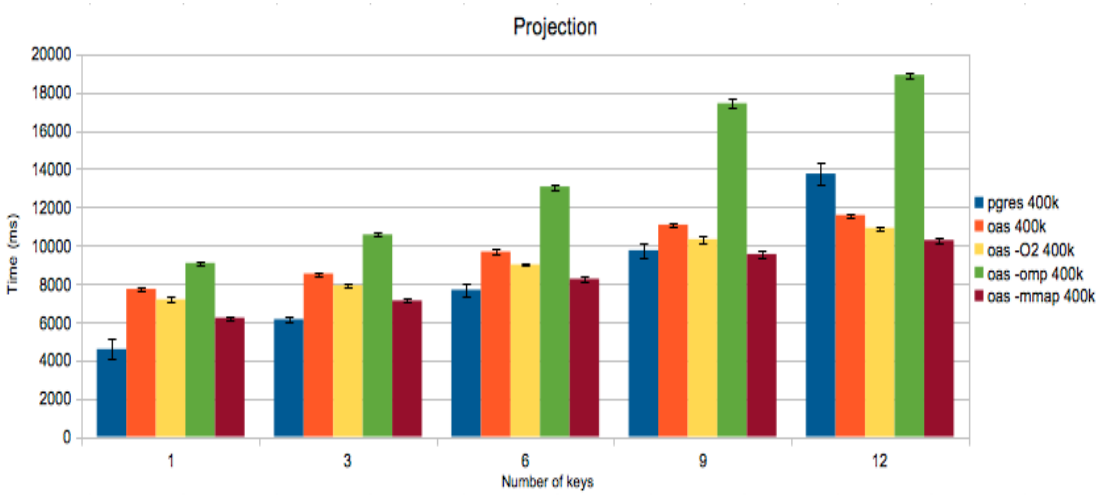

Figure 5.19: Projection times for table size of 400k.

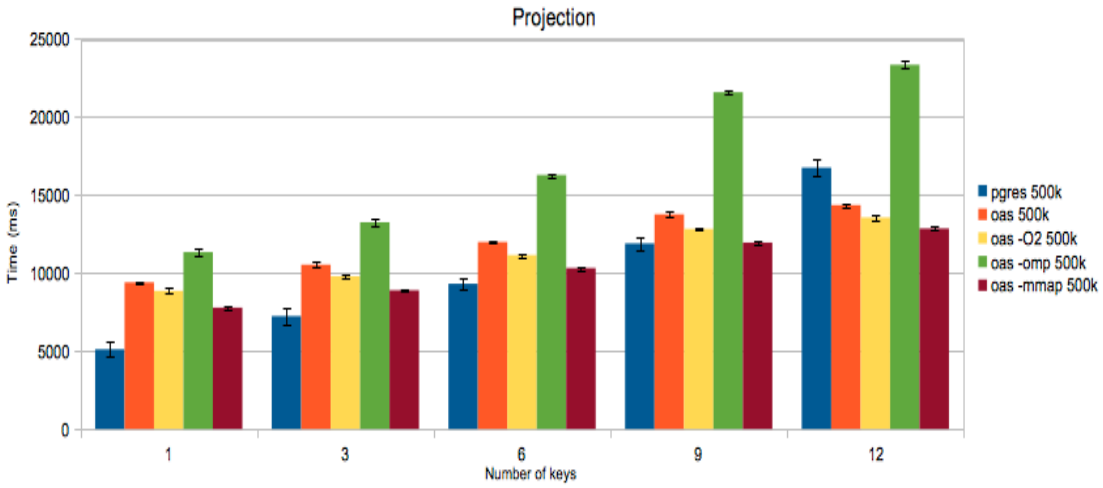

Figure 5.20: Projection times for table size of 500k.

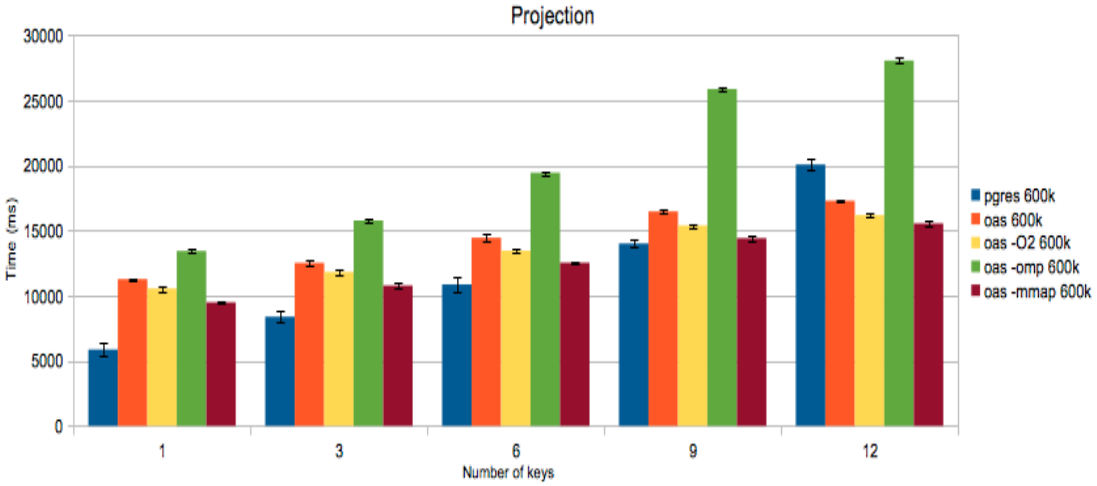

Figure 5.21: Projection times for table size of 600k.

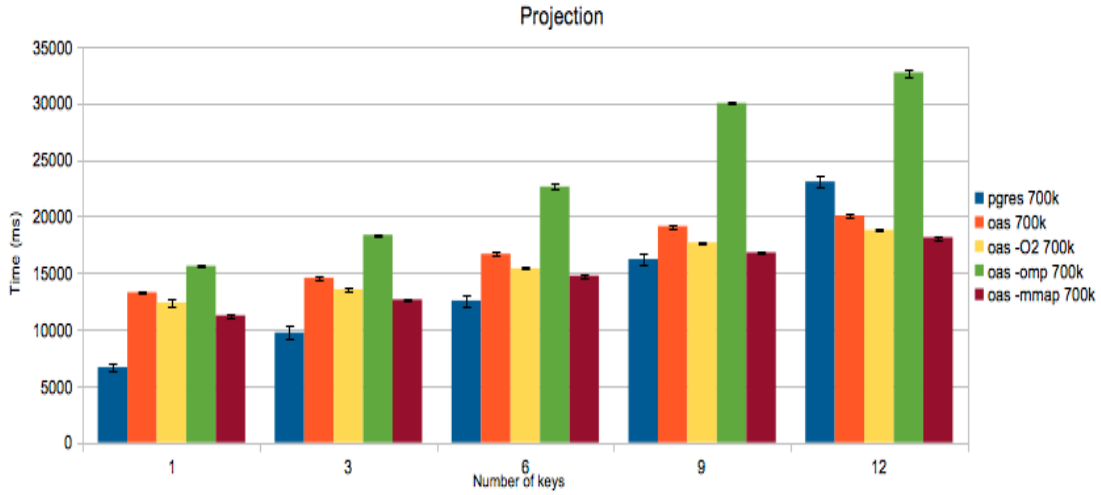

Figure 5.22: Projection times for table size of 700k.

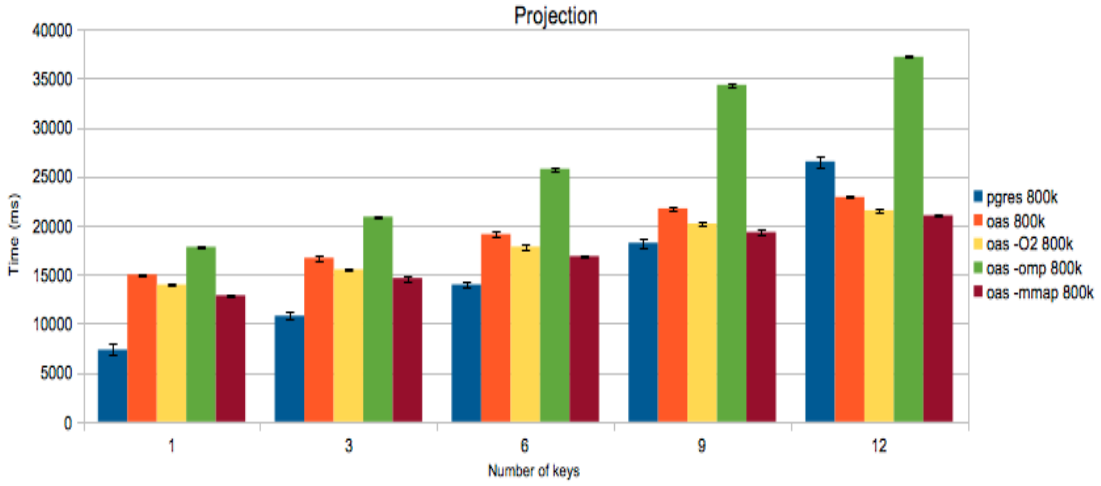

Figure 5.23: Projection times for table size of 800k.

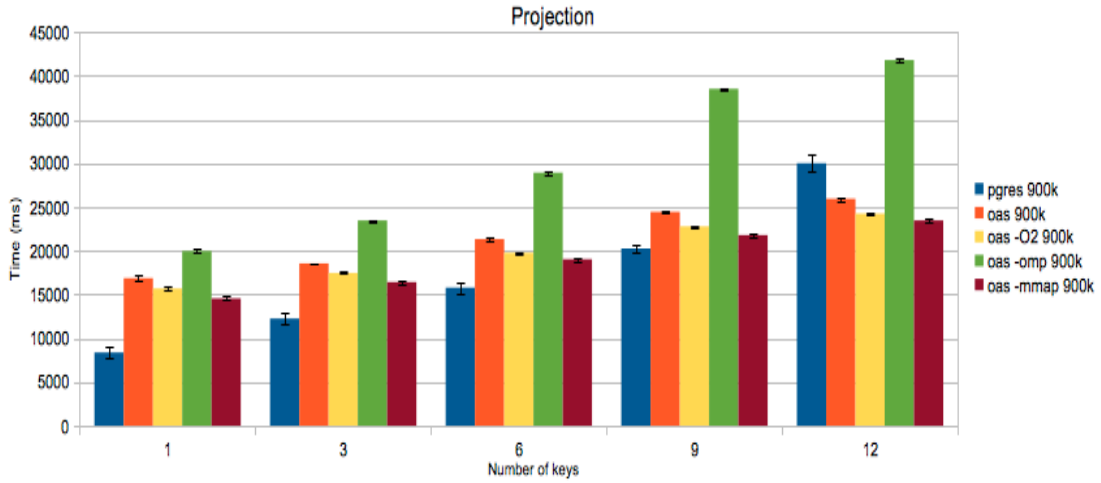

Figure 5.24: Projection times for table size of 900k.

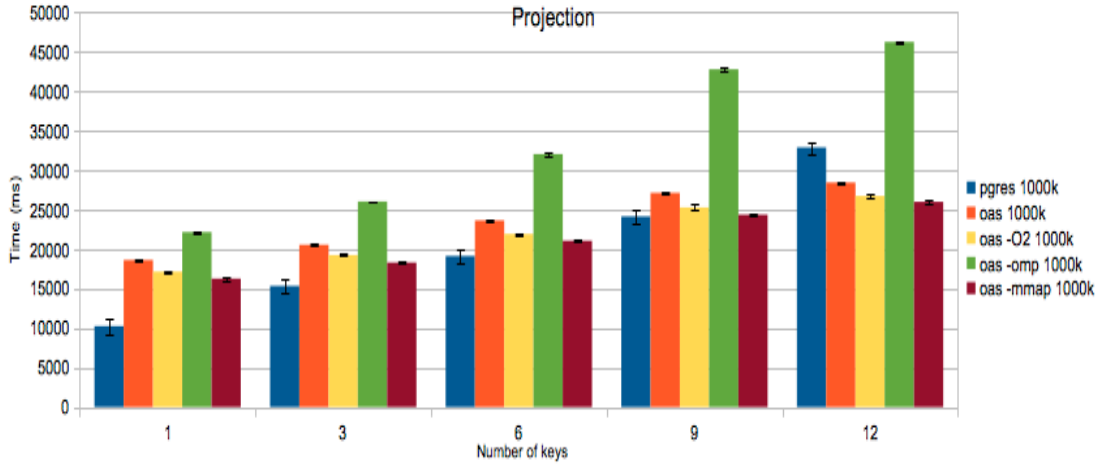

Figure 5.25: Projection times for table size of 1000k.

5.2.4 Sorting

All charts for these queries can be found at the end of this section. These charts show a heavy advantage to the compiled queries over the PostgreSQL queries. In most cases the compiled queries take between half to two thirds of the time to execute as the PostgreSQL queries. All the systems tested also show no real sign of the number of keys having any effect on the time it takes to execute a query.

Sorting, as well as all other database algorithms, is designed with the idea that the entire relation probably can't fit into main memory. This leads to algorithms that are a little more complicated that those that don't hold to that assumption. This effect can be clearly seen in the results presented here. Since this assumption isn't in place for the compiled queries any sorting algorithm can be used to manipulate the data

directly in main memory. Once again the cost in PostgreSQL of performing so many function calls is heavily outperformed by the ability to access data in arrays already in main memory. The benefits of being able to access data like this can be seen in the performance increase for compiled queries.

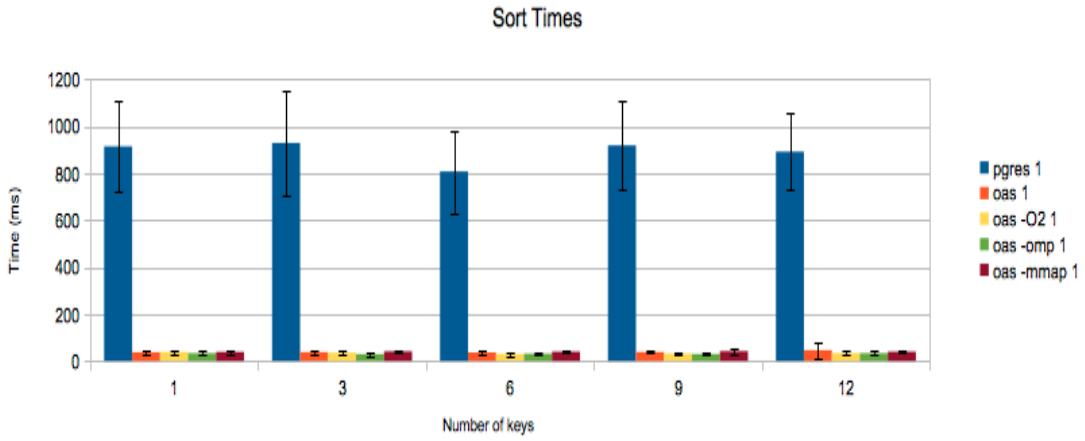

Figure 5.26: Sorting times for table size of 1.

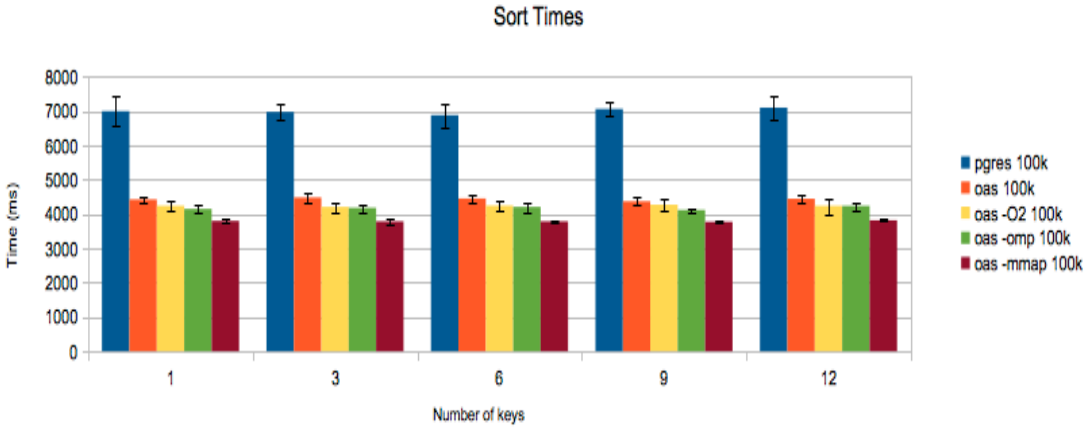

Figure 5.27: Sorting times for table size of 100k.

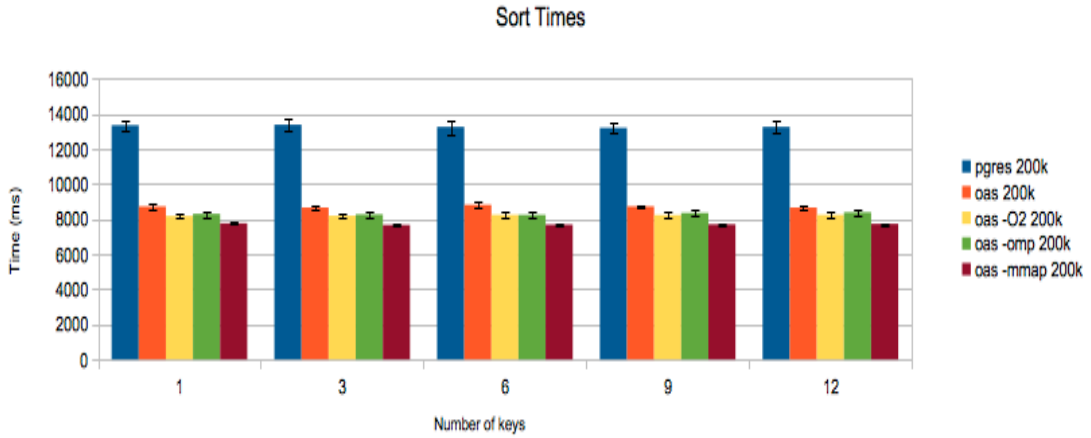

Figure 5.28: Sorting times for table size of 200k.

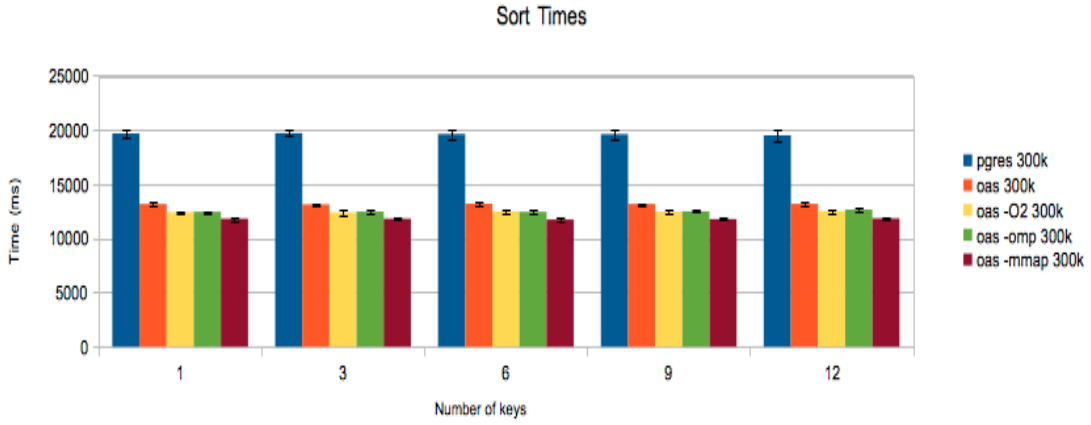

Figure 5.29: Sorting times for table size of 300k.

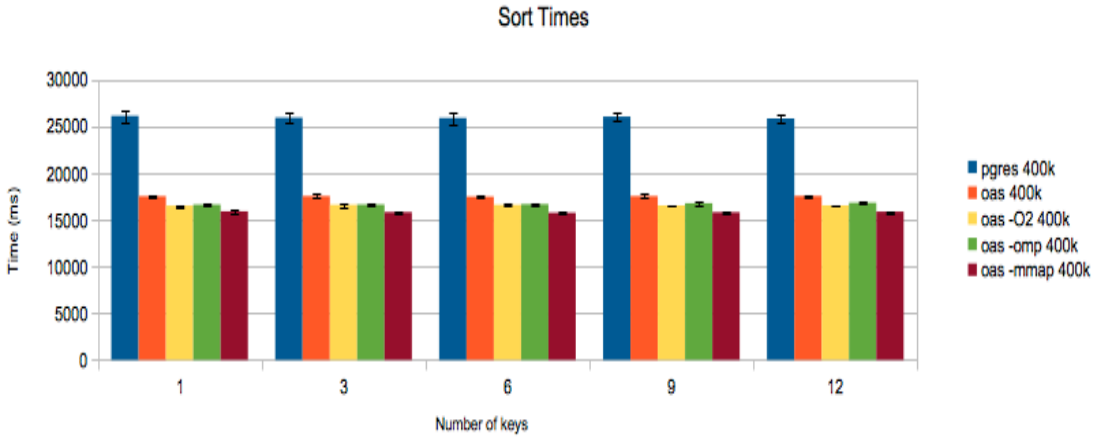

Figure 5.30: Sorting times for table size of 400k.

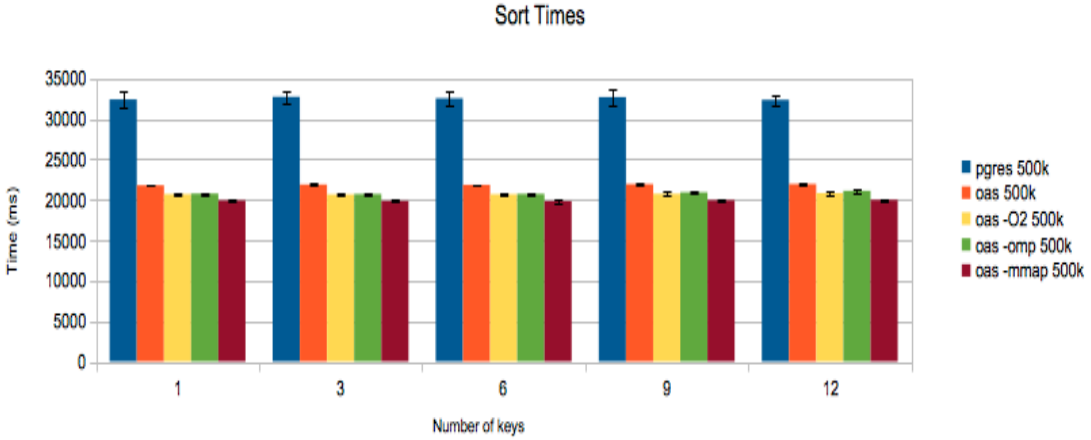

Figure 5.31: Sorting times for table size of 500k.

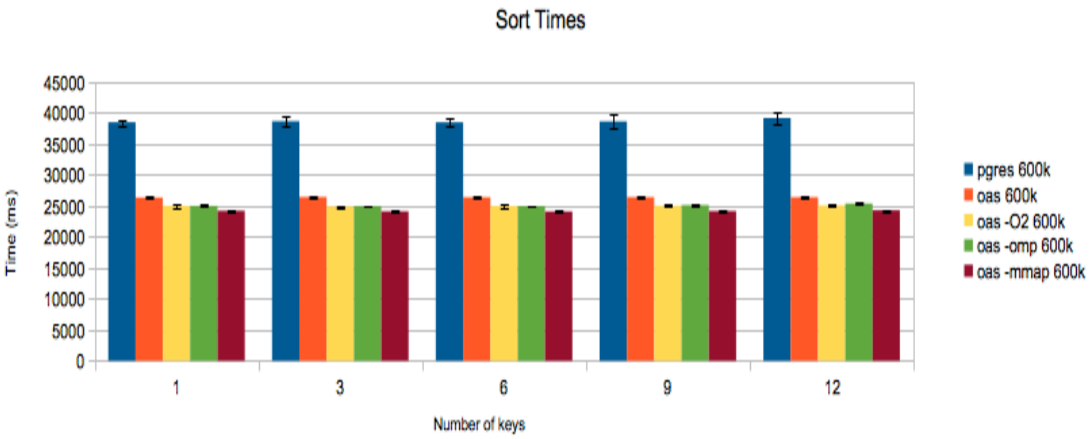

Figure 5.32: Sorting times for table size of 600k.

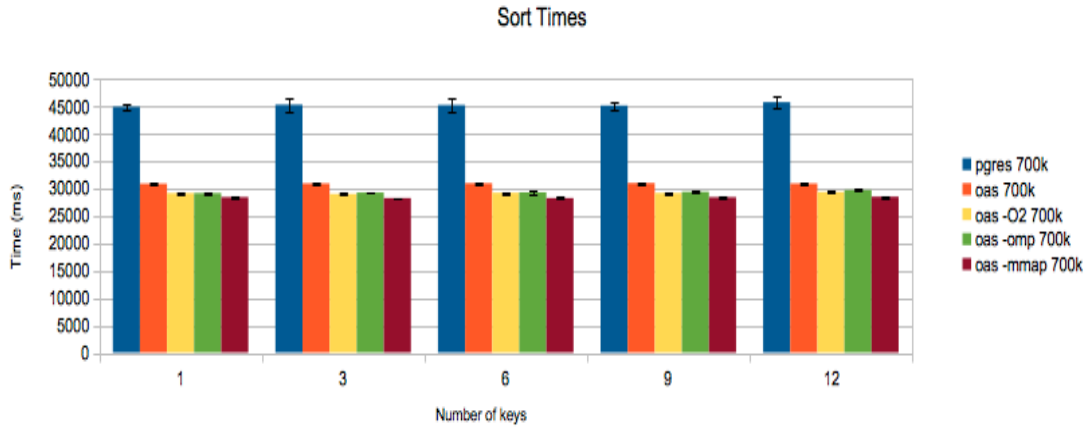

Figure 5.33: Sorting times for table size of 700k.

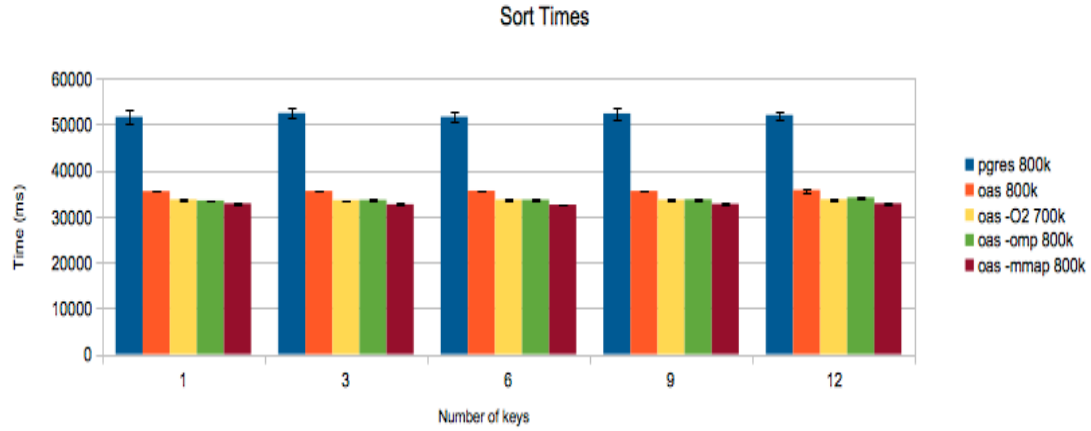

Figure 5.34: Sorting times for table size of 800k.

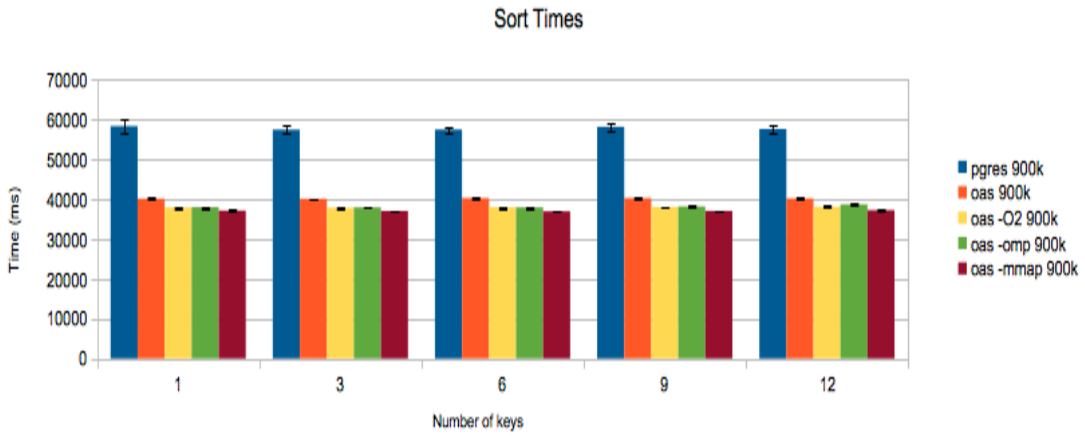

Figure 5.35: Sorting times for table size of 900k.

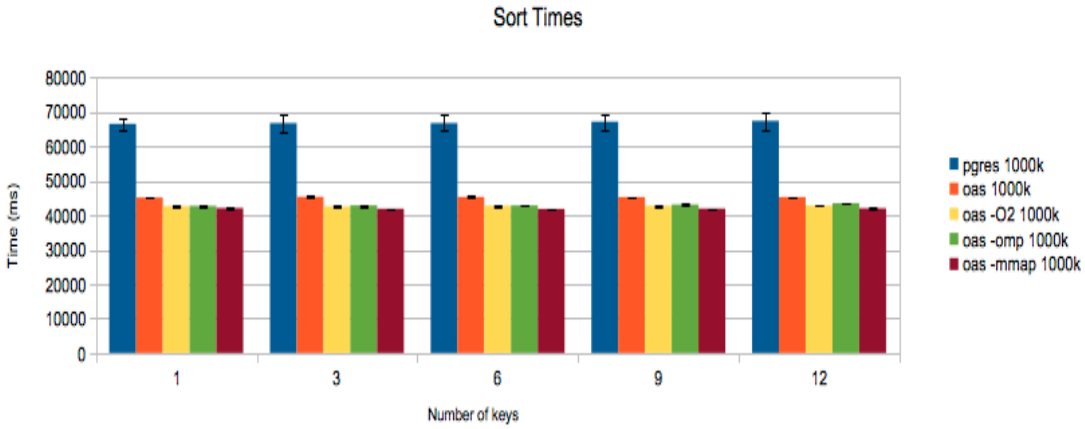

Figure 5.36: Sorting times for table size of 1000k.

5.2.5 Scan, Select, Project, Sort

All charts for these queries can be found at the end of this section. These charts are extremely similar to the projection charts presented earlier. The PostgreSQL system outperforms everything else on table sizes above 100,000 and less then twelve predicates. In addition the difference in execution time between one predicate and twelve on PostgreSQL is very large. Conversely, it can be seen that the compiled queries are much less affected by the number of predicates involved. Due to the fact that the charts are of such similar shape to those from projections is seems safe to assume the projection is the dominating factor in the computation.

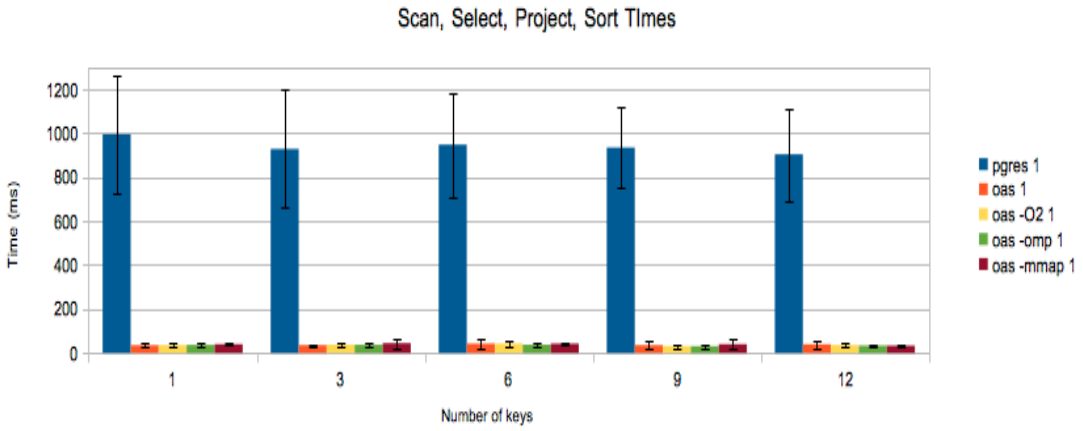

Figure 5.37: Large query times for table size 1.

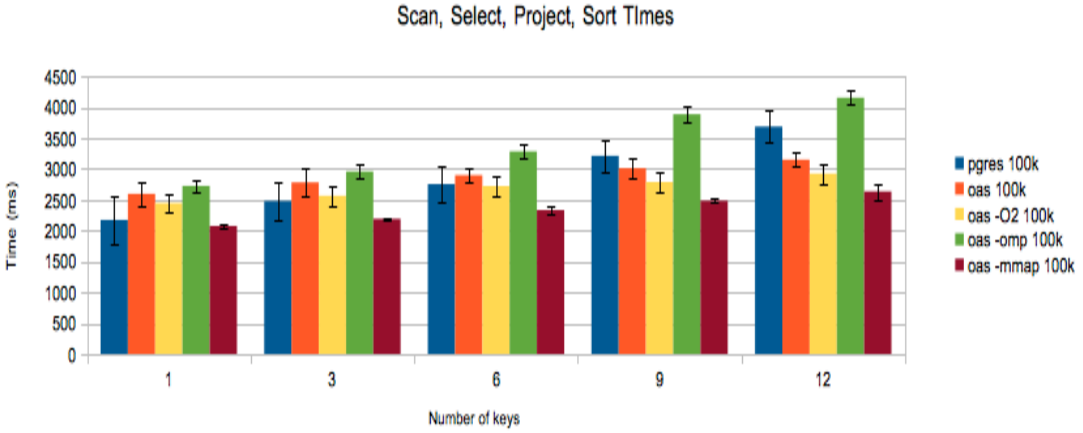

Figure 5.38: Large query times for table size 100k.

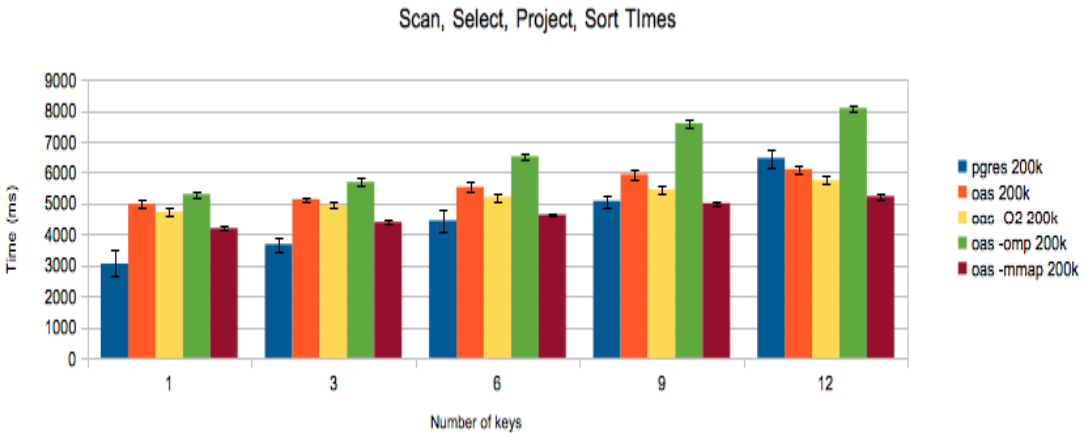

Figure 5.39: Large query times for table size 200k.

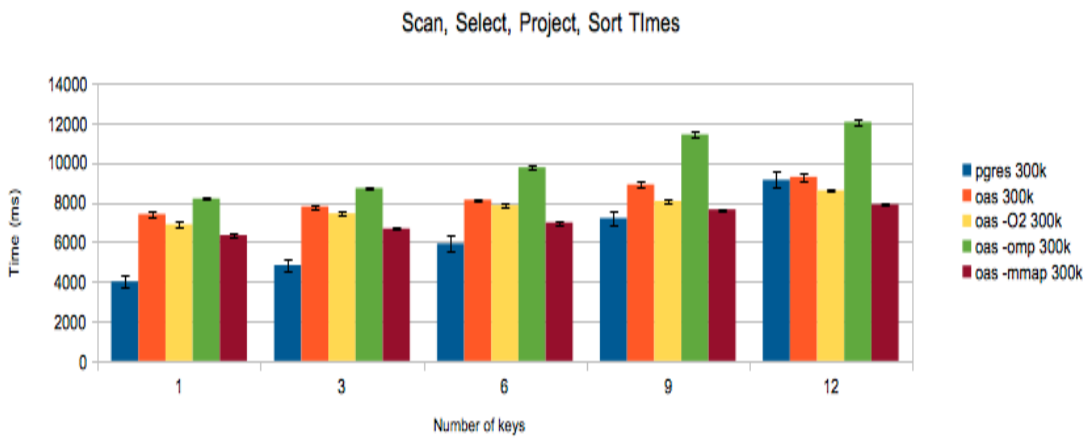

Figure 5.40: Large query times for table size 300k.

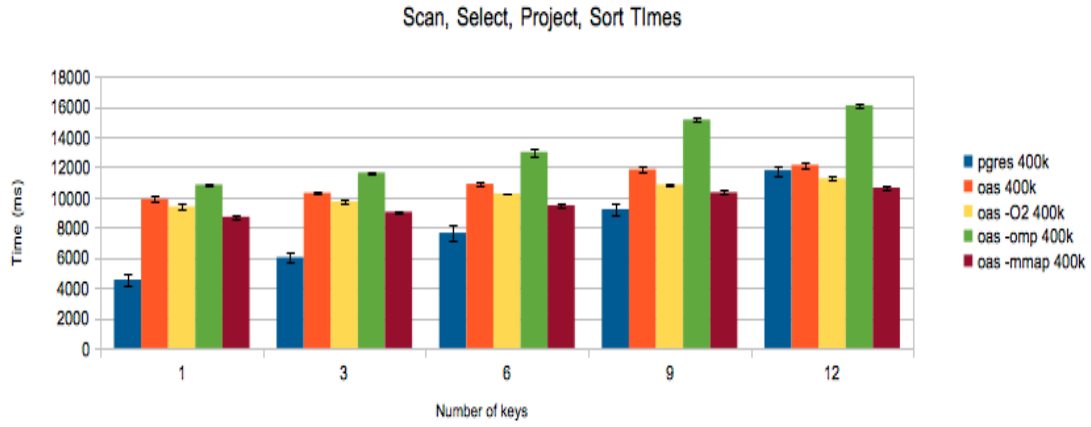

Figure 5.41: Large query times for table size 400k.

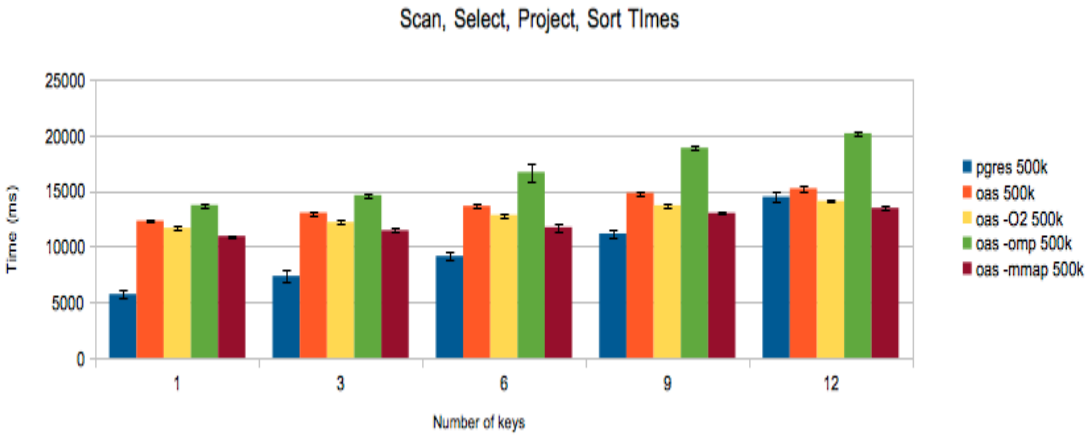

Figure 5.42: Large query times for table size 500k.

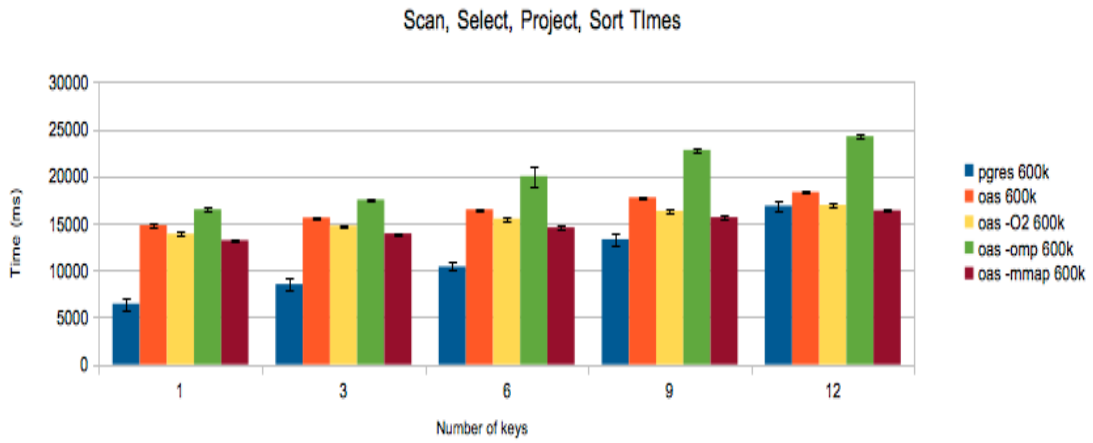

Figure 5.43: Large query times for table size 600k.

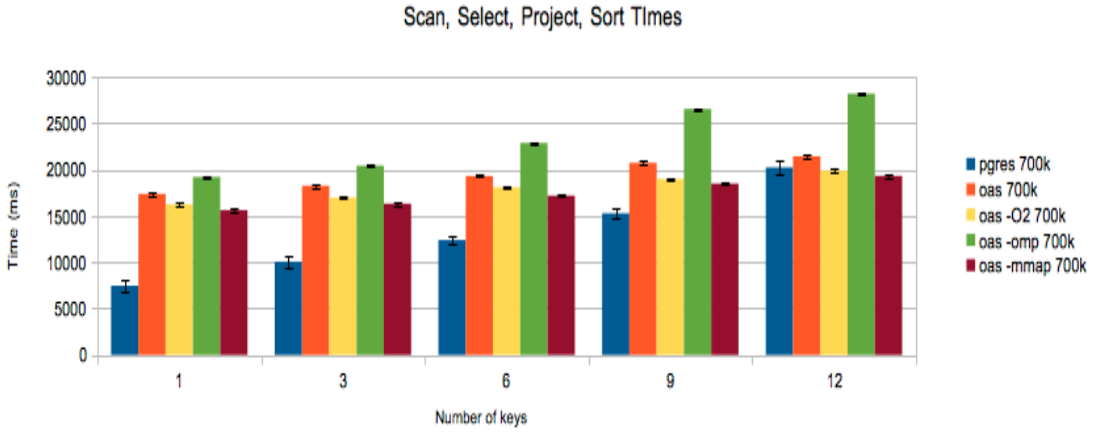

Figure 5.44: Large query times for table size 700k.

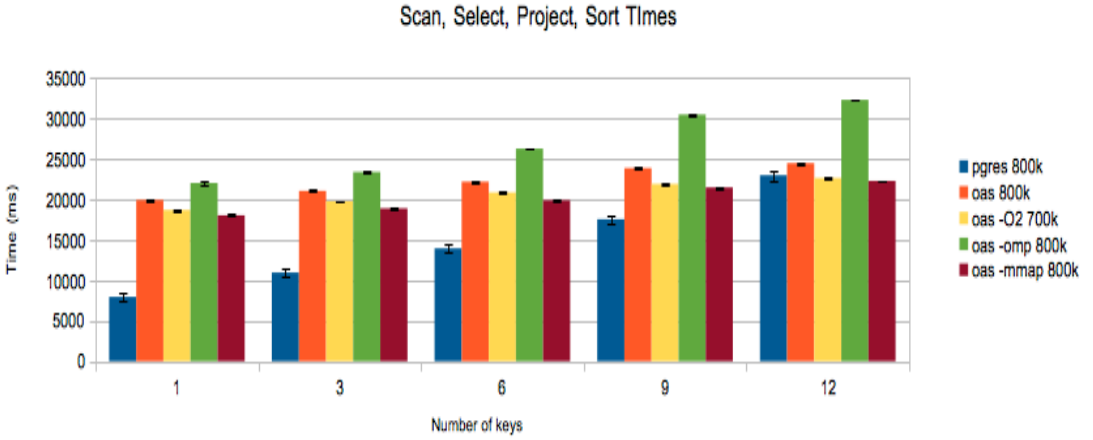

Figure 5.45: Large query times for table size 800k.

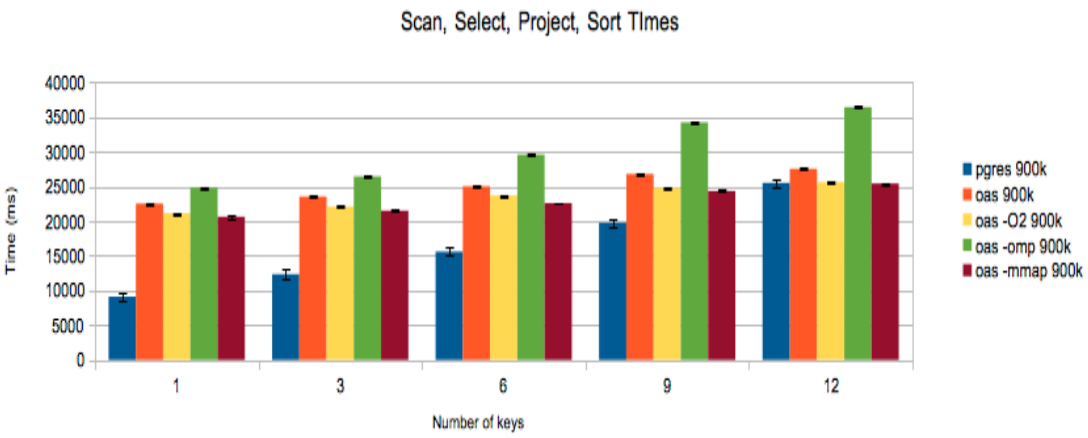

Figure 5.46: Large query times for table size 900k.

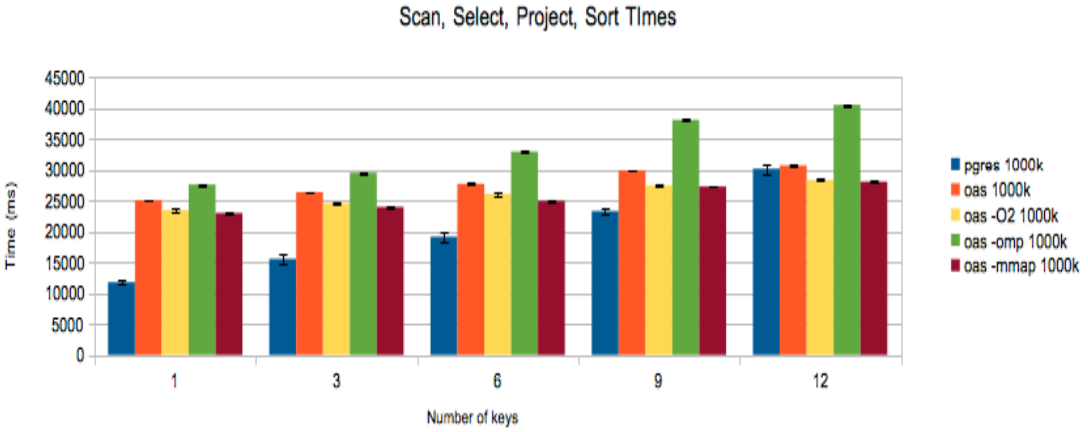

Figure 5.47: Large query times for table size 1000k.

Chapter 6

Conclusion

The goal of this project was to create a system with which database queries could be evaluated without the bloat of a full scale query engine and that would take modern hardware into account during execution. In order to do this it was decided that machine specific executables would be generated that would correspond to various queries. This type of query execution allows for modern processors to be used to their full potential and therefore increase the overall performance of the system. Without the unnecessary bloat of the query engine and having machine specific executables we are able to get some performance increase over a modern query engine.

From examining the data it is clear that compiling queries can have a very positive impact on runtime, especially for smaller table sizes. This can be seen most in the selection and sorting queries. Though the difference is small in some cases it is extremely large in others. The queries that involved only one record are examples of this. It can also be seen that for some reason PostgreSQL is much more effective at processing large tables as well as projections with small amounts of keys. If the results demonstrated for smaller tables could be replicated with larger tables this system could be immensely faster than what PostgreSQL offers. It is also clear from the data presented that the parallelizations used for this project do not fit. This is most likely because the loops that are being parallelized in each case don't contain enough work for each thread to do to justify the cost of thread creation and destruction.

After some profiling of the various compiled versions it was determined that the bulk of time in each query was spent in setting up the relation in memory. This includes setting up the memory for the various data structure and their associated arrays. The majority of time was spent in the setup of each individual record and the data for each field. These memory allocations (and subsequent deallocations) provide for a lot of

the time spent in evaluating each query. One potential reason for this is that all data for each record is stored as a character array. So, whenever a new piece of data is encountered memory corresponding to the length of data needs to be allocated so that the new data can be stored.

Chapter 7

Future Work

The framework developed here has many possible future directions which it could take. One very obvious potential future would be to expand upon the number of operators supported by the compiler to include the other algebraic operations important in query evaluation. Another direction this could be taken is to examine and refine query evaluation on larger files as well as projections to increase the speed of the operators already present. As a somewhat different exercise some research could be done that would examine various compiler optimizations to increase the performance. There is research in the area of trying to find the right compiler optimizations for the given architecture that can be found in [10]. This could prove to be a very interesting project in the examination of various optimizations on this type of work. However, the real future of this project is to create a library of query executables. If a database server is running the same executable over and over this executable could be compiled once and stored. Then whenever that query needs to be executed again all the query engine would need to do would be load and execute that file.

Bibliography

[1] http://www.inf.ed.ac.uk/teaching/courses/adbs/attica/WBGen.java.

[2] Openmp. https://computing.llnl.gov/tutorials/openMP/.

[3] The openmp api specification for parallel programming. http://openmp.org/wp/.

[4] *Purge man page*, September 2005.

[5] *mmap man page*, April 2006.

[6] David J. Dewitt. The wisconsin benchmak: Past, present, future.

[7] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[8] Andi Hellmund. Gcc front-end internals. http://blog.lxgcc.net/wp-content/uploads/2011/03/GCC_frontend.pdf, March 2011.

[9] K. Krikellas, S.D. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 613 –624, march 2010.

[10] Hugh Leather, Michael O'Boyle, and Bruce Worton. Raced profiles: Efficient selection of competing compiler optimizations. June 2009.

[11] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4:539–550, June 2011.

[12] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[13] Jun Rao, H. Pirahesh, C. Mohan, and G. Lohman. Compiled query execution engine using jvm. In *Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on*, page 23, april 2006.

[14] Josh Reese. Gsc - an sql front end for gcc, 2011.

[15] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, DaMoN '11, pages 33–40, New York, NY, USA, 2011. ACM.