University of Warsaw
Faculty of Mathematics, Computer Science and Mechanics

VU University Amsterdam
Faculty of Sciences

Juliusz Sompolski
Student no. 248396(UW), 2128039(VU)

Just-in-time Compilation in Vectorized Query Execution

Master thesis
in COMPUTER SCIENCE

Supervisor:
Marcin Żukowski
VectorWise B.V.

First reader:
Peter Boncz
Dept. of Computer Science,
VU University Amsterdam
Centrum Wiskunde & Informatica

Second reader:
Henri Bal
Dept. of Computer Science,
VU University Amsterdam

August 2011

Supervisor's statement

Hereby I confirm that the present thesis was prepared under my supervision and that it fulfils the requirements for the degree of Master of Computer Science.

Date Supervisor's signature

Author's statement

Hereby I declare that the present thesis was prepared by me and none of its contents was obtained by means that are against the law.

The thesis has never before been a subject of any procedure of obtaining an academic degree.

Moreover, I declare that the present version of the thesis is identical to the attached electronic version.

Date Author's signature

Abstract

Database management systems face high interpretation overhead when supporting flexible declarative query languages such as SQL. One approach to solving this problem is to on-the-fly generate and compile specialized programs for evaluating each query. Another, used by the VectorWise DBMS, is to process data in bigger blocks at once, which allows to amortize interpretation overhead over many database records and enables modern CPUs to e.g. use SIMD instructions. Both of these approaches provide an order of magnitude performance improvement over traditional DBMSes. In this master thesis we ask the question whether combining these two techniques can yield yet significantly better results than using each of them alone. Using a proof of concept system we perform micro benchmarks and identify cases where compilation should be combined with block-wise query execution. We implement a JIT query execution infrastructure in VectorWise DBMS. Finally, using our implementation, we confirm our findings in benchmarks inside the VectorWise system.

Keywords

databases, query execution, JIT compilation, code generation

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatics, Computer Science

Subject classification

H. Software
H.2 DATABASE MANAGEMENT
H.2.4 Systems: Query processing

Contents

Chapter 1

Introduction

This thesis was written at the VectorWise company, part of the Ingres Corporation. The research concentrated around the VectorWise database engine, originating from MonetDB/X100 research project at the CWI in Amsterdam [Zuk09]. The aim was to investigate the possibilities of introducing Just-In-Time (JIT) source code generation and compilation of functions performing query execution, and evaluate and integrate these techniques in the framework of the existing VectorWise query execution engine. On the way, other questions raised and have been answered, concerning various aspects of combining vectorization and compilation in a database engine. This research should be valuable for database systems architects, both from the perspective of extending a vectorized database with query compilation and from the perspective of introducing vectorized processing into a JIT compiled system.

1.1. Motivation

Database systems provide many useful abstractions such as data independence, ACID properties, and the possibility to pose declarative complex ad-hoc queries over large amounts of data. This flexibility implies that a database server has no advance knowledge of the queries until run-time, which leads most systems to implement their query evaluators using an interpretation engine. Such an engine evaluates plans consisting of algebraic *operators*, such as Scan, Join, Project, Aggregation and Select. The operators internally include *expressions* such as boolean conditions used in Joins and Select, calculations used to introduce new columns in Project, and functions like MIN, MAX and SUM used in Aggregation. Most query interpreters follow the so-called iterator-model (as described in Volcano [Gra94]), in which each operator implements an API that consists of open(), next() and close() methods. Each next() call produces one new tuple, and query evaluation follows a "pull" model in which next() is called recursively to traverse the operator tree from the root downwards, with the result tuples being pulled upwards.

It has been observed that the tuple-at-a-time model leads to interpretation overhead: the situation that much more time is spent in evaluating the query plan than in actually calculating the query result. Additionally, this tuple-at-a-time interpretation particularly affects high performance features introduced in modern CPUs [Zuk09]. For instance, the fact that units of actual work are hidden in the stream of interpreting code and function calls, prevents compilers and modern CPUs from getting the benefits of deep CPU pipelining and SIMD instructions, because for these the "work" instructions should be adjacent in the instruction stream and independent of each other [ZNB08, Zuk09].

MonetDB [Bon02] reduced interpretation overhead by using *bulk processing*, where each operator would fully process its input, and only then invoke the next execution stage. This idea has been further improved in the X100 project [BZN05], later evolving into VectorWise, with *vectorized execution*. It is a form of block-oriented query processing [PMAJ01], where the next() method rather than a single tuple produces a block (typically 100-10000) of tuples. In the vectorized model, data is represented as small single-dimensional arrays (vectors), easily accessible for CPUs. The effect is that the percentage of instructions spent in interpretation logic is reduced by a factor equal to the vector-size. It was shown that vectorized execution can improve data-intensive (OLAP) queries performance by a factor 50 over interpreted tuple-at-a-time systems [ZBNH05].

Just-In-Time (JIT) query compilation has been considered as an alternative method for eliminating the ill effects of interpretation overhead. On receiving a query for the first time, the query processor compiles (a part of) the query into a routine that gets subsequently executed. Query compilation removes the interpretation overhead altogether. A compiled code always can be made faster than interpreted code. In analytic workloads with long running queries the compilation time is small compared to query execution time, and the compiled code can be reused, because many queries are repeated with different parameters. Therefore, query compilation has often been considered the ultimate solution to query execution.

In state-of-the-art database engines applying JIT compilation to query execution still fits the tuple-at-a-time model. In place of a tree of interpreted operators, with one tuple being passed using the next() interface, operations are compiled together, but still process one tuple at a time. Such systems observe an order of magnitude improvement in query performance over interpreted tuple-at-a-time systems, just as vectorized systems did.

1.1.1. Research questions

In both cases much of the gains came from removing the interpretation overhead. The research question arising from the perspective of VectorWise is:

- Since a vectorized systems already removed most of this overhead, would it still be beneficial to introduce JIT query compilation to it?

A symmetrical research question can be asked from a perspective of an architect of a database engine applying JIT query compilation:

- Can benefits of vectorization still make an improvement to a tuple-at-a-time compiled system?

When combining vectorization with JIT compilation, the focus has to turn away from the aspect of reducing interpretation overhead, since it is already done with each of these techniques alone. Other potential benefits have to be found and explored.

In vectorized execution, the functions that perform work typically process an array of values in a tight loop. Such tight loops can be optimized well by compilers, e.g. unrolled when beneficial, and, depending on the storage format, enable compilers to generate SIMD instructions automatically. Modern CPUs also do well on such loops because function calls are eliminated, branches get more predictable, and out-of-order execution in CPUs often takes multiple loop iterations into execution concurrently, exploiting the deeply pipelined resources of modern CPUs. A tuple-at-a-time compiled system has just one tuple available at a given time, therefore it does not have the possibility to exploit any inter-tuple micro parallelism using SIMD. It performs many different subsequent operations on this one tuple, which are often dependent of each other. Control and data dependencies hinder the possibilities of deep

CPU pipelining. Therefore, these are the potential reasons why vectorized execution could improve a compiled system.

There are however many operations which are hard to express in a vectorized way. Vectorized execution needs additional materialization of intermediate results – vectors of data have to be written in each basic step. Even though systems like VectorWise go through lengths to ensure that these arrays are CPU cache-resident, this materialization constitutes extra memory load and store work. Compilation can avoid this work by keeping results in CPU registers all the time as they flow from one expression to the other. The memory access pattern of vectorized execution is sometimes sub-optimal, for example when an operation involving multiple columns with values accessed from random positions (e.g. a hash table) has to be executed as many basic vectorized operations, each accessing a vector of these random positions anew. Vectorized execution is anyway better than interpreted tuple-at-a-time execution because of the latter's overhead, but switching to a compiled tuple-at-a-time execution could be beneficial. Just-in-time compilation can be used in a vectorized system to change an operation consisting of multiple vectorized steps into one compiled function, executing a more complex operation in a single loop over tuples.

Introducing both vectorization and compilation into a DBMS also has a software engineering aspect. Implementing additional subsystems makes the system more complex and requires additional maintenance. Vectorization has the advantage of the possibility of very fine grained profiling and error detection. Each basic vectorized function is a big enough block that it can be measured individually during execution – the measurement overhead is small enough compared to an operation working on a whole vector of tuples. This cannot be said about tuple-at-a-time processing – starting and stopping counters around individual operations would be more expensive than the operations themselves. JIT compiling the whole query together makes it hard to later trace which parts of it take how much processing time. Combining vectorization with JIT compilation is a middle ground, because we only generate code for small fragments of query execution, and we can directly compare their performance to that of the vectorized functions they replaced.

1.1.2. Contributions

Combining vectorized processing with JIT compilation is therefore a matter of switching between vector-at-a-time and tuple-at-a-time processing of combined instructions for the sake of optimally utilizing the characteristics of modern CPUs. We identify the situations where it is the case.

We study which operations are better performed in basic vectorized loops, and which would benefit from being compiled into a single loop. Our findings are that in many cases neither pure vectorization nor blind single-loop compilation is the best solution, and the two have to be carefully combined. The results of our work were published as a paper for the DaMoN workshop in June 2011 in Athens [SZB11].

The experiments were made from the perspective of VectorWise, a vectorized system in which we introduce JIT compilation, but the findings can also be applied to introducing vector-at-a-time processing into compiled systems. Until now, in such systems processing was done tuple-at-a-time and intermediate materialization was always treated as an overhead. We show that it is not always true, and switching to vector-at-a-time processing can also improve parts of compiled query execution.

1.2. Outline

The outline of this thesis is as follows:

- Chapter 2 provides an introduction to database systems, focusing on VectorWise DBMS and its aspects relevant to this thesis.
- Chapter 3 evaluates the benefits of introducing Just-in-time query compilation into a vectorized database system and vice-versa. The material from this chapter is the main scientific contribution of this thesis.
- Chapter 4 discusses an infrastructure for Just-in-time compilation, implemented inside the VectorWise database system.
- Chapter 5 describes the implementation of source generation for fragments of query execution in VectorWise.
- Chapter 6 shows an evaluation of the actual implementation that was made in the VectorWise system on TPC-H industrial benchmark. The results are compared to the conclusions of experiments from Chapter 3.
- We conclude and sketch questions and directions for future work in Chapter 7.

Chapter 2

Preliminaries

This chapter introduces concepts from the field of database systems that will be used throughout this thesis. It starts with general databases introduction in Section 2.1, followed by an introduction to the VectorWise DBMS architecture and vectorized query execution model in Section 2.2. Implementation details that will be relevant to the task of integrating this DBMS with JIT query compilation infrastructure are highlighted. The reasoning behind many of the methods used in this thesis lies in the hardware architecture, so an introduction to modern hardware is provided in Section 2.3. An overview of what has to be done to implement JIT query compilation and previous research and existing systems that already apply JIT are described in Section 2.4.

2.1. Relational database systems

A *database management system* (DBMS) is a software package of programs that provide services for users and applications to store and access data. It allows multiple concurrent users to query and modify data using a high level query language such as SQL.

The *relational database management system* (RDBMS, we will often use just DBMS) is the most common type of a DBMS. It uses relations as data model. A relation is a set of tuples, best viewed as a table, with rows being *tuples* and columns being *attributes*. In contrary to mathematical relations, the attributes are named and typed.

Database

A *database* is a collection of relations/tables stored and managed by the DBMS. The database is described by a schema. The schema specifies what tables are in the database, what attributes are in each table, and how the tables reference one another. An example of database tables is shown in Figure 2.1.

Structured Query Language and relational algebra

Structured Query Language (SQL) can be used to express querying operations on a database. It enables selecting tuples fulfilling given conditions, joining data from multiple relations together, calculating derived values and aggregates grouped by certain attributes. SQL has understandable syntax, resembling a querying statement sentence in English language.

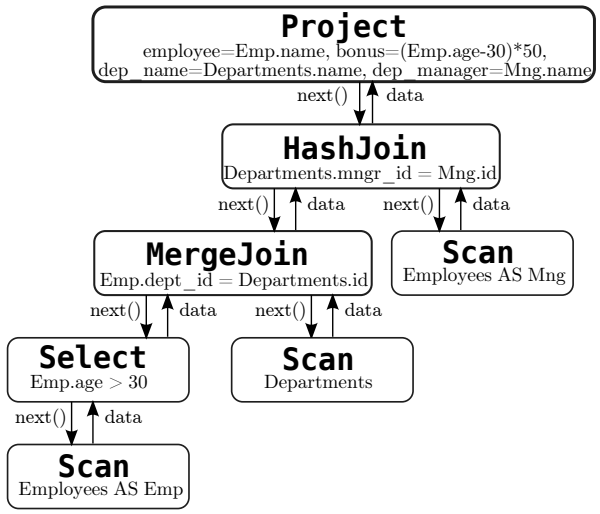
SQL queries correspond to relational algebra. The algebra consists of operators, such as:

Employees			
id	*name*	*age*	*dept_id*
1	Adam	39	1
2	Eva	25	1
3	Tom	31	1
4	Mark	60	2
5	Chris	42	2

Departments		
id	*name*	*mngr_id*
1	Production	1
2	Sales	4

```
SELECT Emp.name AS employee, bonus = (Emp.age - 30) * 50
       Departments.name AS dep_name,Mng.name AS dep_manager,
FROM Employees AS Emp
JOIN Departments      ON Emp.dept_id = Departments.id
JOIN Employees AS Mng ON Mng.id = Departments.mngr_id
WHERE Emp.age > 30
```

$$\Pi_{\substack{employee=Emp.name,\\ bonus=(Emp.age-30)*50,\\ dep\_name=Departments.name,\\ dep\_manager=Mng.name}} \left(\begin{array}{c} (\sigma_{Emp.age>30}(\rho_{Emp/Employees}(Employees)) \bowtie_{\substack{Emp.dept\_id=\\ Departments.id}} \\ Departments) \bowtie_{\substack{Mng.id=\\ Departments.mngr\_id}} \rho_{Mng/Employees}(Employees) \end{array} \right)$$



Query result			
employee	*bonus*	*dep_name*	*dep_manager*
Adam	450	Production	Adam
Tom	50	Production	Adam
Mark	1500	Sales	Mark
Chris	600	Sales	Mark

Figure 2.1: Example of database table, SQL query, the corresponding operations in relational algebra, and a Volcano model operator tree.

- *Projection* (Π) and *Rename* (ρ), which are used to choose and maintain the subset of attributes to be passed on, as well as to introduce new attributes derived from the existing ones by e.g. arithmetic calculations.
- *Selection* (σ), which is used to select tuples that match a certain boolean condition.
- *Join* ($\bowtie$), used to combine tuples from two tables (*left* and *right*) using a given condition. The condition is most commonly an equality of an attribute from the left table to an attribute from the right table (*equi-join*). There are many different types of join, depending on whether the result should contain only tuples matched from both operands (*inner join*), or should unmatched tuples from left, right or both tables be also produced,

extended with empty (NULL) values on the unmatched side (*left*, *right* or *full outer join*).

- *Aggregation* ($\mathcal{G}$), used to aggregate a function such as sum, average, minimum or maximum of an attribute or count of tuples, grouped by the value of another attribute(s).

An example of a SQL query and corresponding relational algebra expression is shown in Figure 2.1.

Query optimization

A parsed SQL query in a database system resembles a tree of relational algebra operators. It is the work of query optimizer to match these operators with their implementations in the system. There may be multiple concrete operators for a given abstract algebraic operator, e.g. a join can be implemented either as a HashJoin, with one relation put in a hash table, or a MergeJoin, when both relations were stored sorted on the join attributes. Different join algorithms can also be used if we know whether to except at most one match (Join01), exactly one match (Join1) or more matches (JoinN) per tuple from one of the relations. The optimizer needs to decide which of the implementations to use based on the metadata, statistics and value distribution it collects about the data.

For example, in Figure 2.1 MergeJoin was used for one of the joins, because data was sorted on the department id, and HashJoin was used for the other join.

Volcano operator model

Concrete operators in the DBMS are usually implemented following the model first introduced in Volcano [Gra94]. The operators form a tree and implement an iterator API, with open(), close() and next() methods. The open() and close() methods are used for initialization and finalization. The next() method makes the operator produce output. In most database systems this is done tuple-at-a-time: exactly one tuple of output will be returned on one call to the next() method.

Data is pulled by the operators upwards in the tree. The root operator produces query output, calling next() on its children, which in turn call their children etc., down until the Scan operators at the leaves of query execution tree, which interact with the storage layer and read the data from the database.

For example, when the next() method would be called on a Project operator, it would call the next() method on its child once, compute the projections on the delivered tuple, and return the result. A Select operator would call next() on its child possibly multiple times, evaluating the filtering condition until a tuple from the child satisfies it and can be returned. More complex operators follow even more complicated logic. When the next() method of a HashJoin with a memory fitting hash table is called for the first time, it would iterate through all output of the child operator of the smaller of joined relations and put all resulting tuples into the hash table. Then upon subsequent next() calls it would call the next() method of the bigger of joined relations and try to match it against tuples in the hash table. There are many variants of when and what actually will be returned, depending on whether it is an inner or outer join and “01”, “1”, or “N” join.

An example Volcano-style operator tree is presented in Figure 2.1.

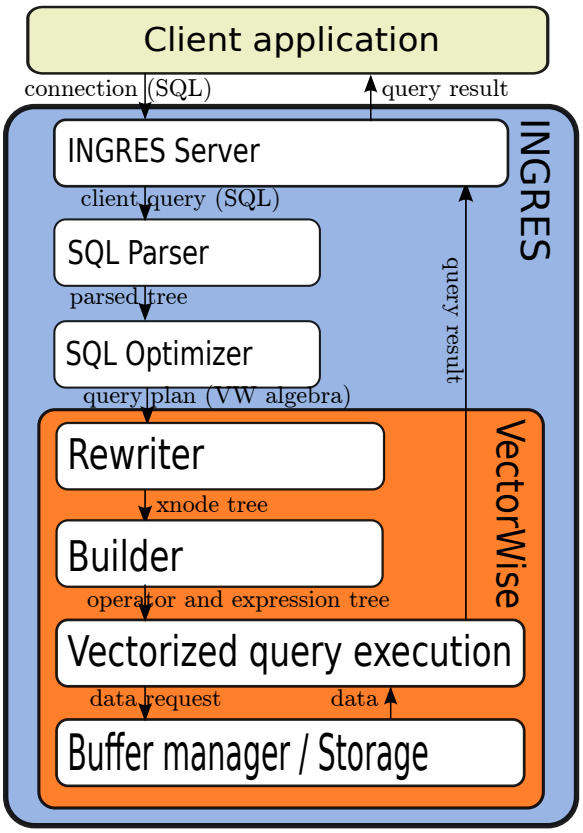


Figure 2.2: Architecture of the Ingres VectorWise system.

2.2. VectorWise vectorized execution

VectorWise [Vec] is a database management system mostly intended for analytical workloads in applications such as online analytical processing, business intelligence, decision support systems and data mining. Such a workload is characterized by being read-intensive, having complex, compute intensive queries and database tables with a lot of attributes. Queries may often only access a few attributes, but touch a large fraction of all rows. VectorWise roots from the MonetDB/X100 project, developed at CWI in The Netherlands [BZN05, Zuk09].

2.2.1. Architecture of Ingres VectorWise system

VectorWise is integrated with Ingres DBMS. Ingres is responsible for the top layers of the database management system stack, while VectorWise implements the bottom layers – query execution, buffer manager and storage. The architecture is shown in Figure 2.2.

The Ingres server maintains connections with users and parses their SQL queries. Ingres keeps track of all schemas, metadata and statistics (e.g. cardinality, distribution) and based on that decides on an optimal query execution plan.

VectorWise algebra

The plan generated by the Ingres optimizer is a query execution tree of algebraic operators implemented in VectorWise, expressed in VectorWise algebra language. Its syntax is simple and can be seen in Figures 2.3 and 2.4. We describe the operators that are used in this thesis:

```
Aggr (
   Project (
      MScan (
          '_lineitem',
             [ '_l_extendedprice',
               '_l_tax',
               '_l_returnflag'
             ]
      ), [ resprice =
             *(_lineitem._l_extendedprice,
               +(sint('1'),_lineitem._l_tax)),
           flag = _lineitem._l_returnflag
          ]
   ), [ flag ],
      [ sumprice = sum(resprice) ]
);
```

```
<AGGR>:
|_<PROJECT>:
| |_<MSCAN>:
| | |_<STRING>: str('_lineitem')
| | |_<LIST>:
| | | |_<STRING>: str('_l_extendedprice')
| | | |_<STRING>: str('_l_tax')
| | | \_<STRING>: str('_l_returnflag')
| | \_<LIST>:
| \_<LIST>:
|   |_<ASSIGNMENT>: resprice
|   | \_<OPERATOR>: str('*')
|   |   |_<IDENT>: _lineitem._l_extendedprice
|   |   \_<OPERATOR>: str('+')
|   |     |_<CALL>: str('sint')
|   |     | \_<STRING>: str('1')
|   |     \_<IDENT>: _lineitem._l_tax
|   \_<ASSIGNMENT>: flag
|     \_<IDENT>: _lineitem._l_returnflag
|_<LIST>:
| \_<IDENT>: flag
|_<LIST>:
  \_<ASSIGNMENT>: sumprice
    \_<AGGROPERATOR>: str('sum')
      \_<IDENT>: resprice
```

Figure 2.3: An expression in VectorWise algebra and the corresponding xnode tree, as pretty-printed from the system log. The xnode tree corresponds to the parsed algebra.

- **MScan**. The MScan operator is the bridge between the storage layer and query execution. It is situated at the leaves of the query tree and it is responsible for providing data read from the database. Its parameters are a table name and a list of columns to be read. As a result of a call to its `next()` method it produces a vector of values from each of the requested attributes in the given table.

- **Select**. The Select operator corresponds to a relational algebra Selection. It takes the input operator and a boolean filtering condition as parameters. A call to its `next()` method produces a vector of indexes in the input vectors for which the condition evaluated to true (*selection vector*, see Section 2.2.2 for a description of how execution is performed with selection vectors).

- **Project**. The Project operator corresponds to a relational algebra Projection and Rename. Its parameters are the input operator and a list of projection expressions. The `next()` method produces vectors with the results of these expressions.

- **Aggr**. The Aggr operator corresponds to a relational algebra Aggregation. It takes an input operator, list of key attributes and a list of aggregate functions. If the list of keys is empty, it performs aggregation into one global group. If the list of aggregate functions is empty, Aggr does duplicate elimination. A call to the `next()` method produces vectors for key attributes, and vectors with the results of aggregation for the corresponding group. Internally the Aggr operator can choose to work either directly using the values of keys as group identifiers when their domain is small enough (DirectAggr), or a hash table otherwise (HashAggr). If the input is sorted on the key attributes, we can also perform ordered aggregation, where the aggregates are computed one group at a time, because all values from one group will come sequentially (OrdAggr). The operator can

decide which method to use based on the metadata.

- **HashJoin family**. HashJoin is a Join relational operator implementation using a hash table to store the smaller of the two joined relations (*build relation*). There is a whole family of HashJoin operators, such as `HashJoinN`, `HashJoin1`, `HashJoin01`, with usage depending on the expected cardinality of matches on both sides. It takes two input operators for the two joined relations, from each of them a list of keys to be matched and a list of other attributes to project, and a flag specifying the matching behaviour (inner or outer, left, right or full). The `next()` method produces vectors of values from the bigger relation (*probe relation*) and the corresponding vectors of matching values from the build relation.
- **MergeJoin family**. MergeJoin family of operators is an alternative family of Join relational operators used when both input relations are sorted on the key attributes, and the join can be performed by simultaneously progressing in both inputs and producing matches in a merging process.

Rewriter

The *Rewriter* performs VectorWise specific semantic analysis on the query plan. It assigns datatypes and other helpful metadata. It also does some optimizations that were not performed by Ingres – e.g. introduces parallelism, rewrites conditions to use lazy evaluation and eliminates dead code. The Rewriter is implemented as a set of rules that are applied to a tree of `xnode` structures, which represent the plan. Figure 2.3 shows an example tree of `xnode`s created from a VectorWise algebra expression. Each Rewriter rule traverses the `xnode` tree in a top-down or bottom-up walk and either transforms it or annotates the nodes with additional meta information (called *properties*). The Rewriter is easily extensible with new rules. More information on Rewriter can be found in Section 5.2 where we describe the new Rewriter rules introduced by JIT compilation and their limitations.

Builder

The *Builder* constructs the actual query execution structures out of the annotated `xnode` tree. These structures are *Operators* and *Expressions*. Operators in VectorWise correspond to the algebraic operators of VectorWise algebra, like Select, Project, Aggr etc. They form a tree implementing the Volcano model iterator API.

Expressions perform the actual basic operations. For example, they do the arithmetic operations in Project operator, evaluate the conditions in Select operator, compute hash values etc. Expressions often do not have corresponding nodes in the `xnode` tree. They form DAGs rather than trees and the logic of what expressions are needed and how they are created and connected is operator specific, implemented in the operator builder. Therefore, the annotated `xnode` tree after the Rewriter passes does not contain all the information and does not correspond exactly to the final query execution tree.

Since expressions are the structures that perform the actual work, they will be the object of JIT code generation in this thesis. Therefore, we will provide some more detailed introduction to expressions in in Section 2.2.2.

Vectorized query execution

When data is processed and transferred one tuple at a time, instructions related to query interpretation and tuple manipulation outnumber the instructions for actual operation. Moreover,

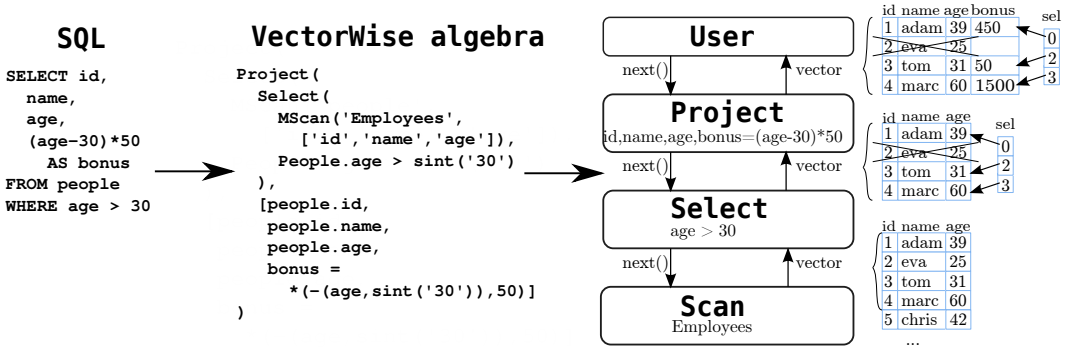


Figure 2.4: Example of translation from SQL query into VectorWise algebra and then a tree of vectorized operators.

generic operator interfaces require virtual function calls, which further reduce performance.

The difference between VectorWise execution and most other databases implementing the Volcano model is that the next() method of the iterator API returns a vector of results instead of a single tuple. Each call to next() method produces a block of 100-10000 tuples (default: 1024). This reduces the interpretation overhead of the Volcano model, because the interpretation instructions are amortized across multiple tuples, and the actual computations are performed in tight loops.

Vectorized processing will be described in more details in Section 2.2.2.

Column storage and vectors

Most database systems use row-wise storage, with all attributes from one tuple stored together, called N-ary Storage Model (NSM). Data in the VectorWise database is stored column-wise. One model is called Decomposition Storage Model (DSM) [CK85], in which a disk page contains consecutive values belonging to just one column of a certain table. A middle ground between DSM and NSM that can also be used in VectorWise is a model called Partition Attributes Across (PAX). In this model a disk page stores data from all columns of a table (i.e. complete tuples), but within the page data is stored column-wise.

DSM has several advantages for analytical workloads. As database tables have many attributes, often only few of which are used in a given query, reading only the needed columns instead of whole table rows greatly reduces the amount of transferred data. Data from a single column can also be compressed efficiently and easily using simple compression schemes such as dictionary encoding or run-length encoding, as the data often comes from a limited or clustered domains. This thesis focuses on query execution and is mostly indifferent to the storage layer. More details about VectorWise storage can be found in [Zuk09].

During execution data is also processed in a column-wise way. Expressions operate on vectors, which are arrays that contain data from a single column. Vectors are allocated by the Builder and data is transferred between operations in them.

2.2.2. Expressions and primitives

In the query execution layer of VectorWise, the actual worker functions are called *primitives*. They are short functions processing arrays of values in a tight loop. They usually perform just one operation, such as an arithmetic calculation, evaluation of a boolean condition, fetching from memory locations pointed to by a vector of indexes etc. They take a number of vectors

Algorithm 1 Examples of templates of *primitive* functions. The *map primitives* perform calculations on input vectors. The *selection primitives* evaluate a boolean condition on the input vectors and write the positions of tuples for which the condition was true to a resulting *selection vector*. Any primitive can work with or without an input selection vector. If a selection vector is present, the calculation is performed only on tuples from selected positions. A NULL selection vector means that the operation shall be performed on all elements of the vectors. They return the number of elements they written to the resulting vector.

```
// Template of a binary map primitive.
uidx map_OP_T_col_T_col(uidx n, T* res, T* col1, T* col2, uidx *sel) {
    if(sel) {
        for(uidx i = 0; i < n; i++) res[sel[i]] = OP(col1[sel[i]],col2[sel[i]]);
    } else {
        for(uidx i = 0; i < n; i++) res[i] = OP(col1[j],col2[j]);
    }
    return n;
}

// Template of a binary map primitive with first parameter being a constant.
uidx map_OP_T_val_T_col(uidx n, T* res, T* val1, T* col2, uidx *sel) {
    if(sel) {
        for(uidx i = 0; i < n; i++) res[sel[i]] = OP(*val1,col2[sel[i]]);
    } else {
        for(uidx i = 0; i < n; i++) res[i] = OP(*val1,col2[j]);
    }
    return n;
}

// Template of an unary map primitive.
uidx map_OP_T_col(uidx n, T* res, T* col1, uidx *sel) {
    if(sel) {
        for(uidx i = 0; i < n; i++) res[sel[i]] = OP(col1[sel[i]]);
    } else {
        for(uidx i = 0; i < n; i++) res[i] = OP(col1[j]);
    }
    return n;
}

// Template of a binary selection primitive.
uidx sel_COND_T_col_T_col_branching(uidx n, uidx *res_sel, T* col1, T* col2, uidx *sel) {
    uidx ret;
    if(sel) {
        for(uidx i = 0; i < n; i++)
            if(COND(col1[sel[i]], col2[sel[i]])) res_sel[ret++] = sel[i];
    } else {
        for(uidx i = 0; i < n; i++)
            if(COND(col1[i], col2[i])) res_sel[ret++] = i;
    }
    return ret;
}

// Alternative template of a binary selection primitive.
uidx sel_COND_T_col_T_col_datadep(uidx n, uidx *res_sel, T* col1, T* col2, uidx *sel) {
    if(sel) {
        for(uidx i = 0; i < n; i++) {
            res_sel[ret] = sel[i]; ret += COND(col1[sel[i]], col2[sel[i]]);
        }
    } else {
        for(uidx i = 0; i < n; i++) {
            res_sel[ret] = i; ret += COND(col1[i], col2[i]);
        }
    }
    return ret;
}
```

(or scalars) as parameters and output a vector of results, which in turn can be the input of another operation. They are arranged into a tree (or DAG) of structures called *expressions* and evaluated in left-deep recursive order, with output from the child expressions fed as input to the parents. The way how expressions are linked together and evaluated is controlled by *operators*, which use them according to their more complex logic. The DAG of expressions makes composite operations out of the basic primitives. The vectors that are being passed between the expressions introduce some materialization overhead, but it is minor because the size of the vector is such that the currently processed vectors fit in the CPU cache.

Algorithm 1 shows examples of code templates of different types of primitives. A `_col` suffix in the primitive name indicates a vector (columnar) parameter, while a `_val` suffix is used for a constant parameter (scalar). Type `T` can be substituted with different types, with `uchr`, `usht`, `uint`, `ulng` representing internal types for 1-, 2-, 4- and 8-byte unsigned integers, and `schr`, `ssht`, `sint`, `slng` being their signed counterparts. `uidx` is an unsigned integer type used for indexes and positions. Each primitive can either work on all elements of the input vectors, or only on selected positions, provided in an input *selection vector*. This way VectorWise handles selections: a `Select` operator only has to output a vector of selected positions in a flow-through stream of tuples, instead of copying the selected values from all columns of a table into new densely-filled vectors. Of course, if the selection vectors become short, so the operations are performed on only few tuples in a block, a special operator called Chunk can be used, which packs the sparse values into new vectors, dropping the need for a selection vector and allowing better amortization of interpretation overhead up in the query plan. There may be more than one primitive implementation for a single operation, for example the two different templates for selection primitives in Algorithm 1. Different versions of a primitive might be chosen depending on some additional logic in the operator.

VectorWise pre-generates primitives for all needed combinations of operations, types and parameter modes. All functions to support SQL fit in ca. 9000 lines of macro-code, which expands into roughly 3000 different primitive functions producing 200K lines of code and a 5MB binary.

2.2.3. VectorWise performance

VectorWise achieves very high performance, with over 50 times speedup over interpreted tuple-at-a-time database systems [ZBNH05]. On one side, it is the effect of optimized storage, which is able to deliver data for execution at a very high rate thanks to its column oriented architecture, light-weight compression schemes and in-cache decompression (more info can be found in [Zuk09]). In this thesis we concentrate more on the execution layer. Thanks to vectorization we get benefits such as amortizing the interpretation overhead over multiple tuples and better possibilities of SIMD processing using SSE.

VectorWise performance is confirmed by the TPC-H industrial benchmark [Tra02]. TPC-H is an ad-hoc, decision support benchmark. The VectorWise DBMS is currently the fastest non-clustered database management system benchmarked in the categories of 100 GB, 300 GB and 1 TB databases.

With introducing JIT compilation we try to improve the performance even further.

2.3. Modern hardware architecture

The reasoning behind many of the methods used in this thesis lies in the architecture of modern hardware. We describe some of the features of modern CPUs that we take advantage from. Detailed information is taken from [Fog11].

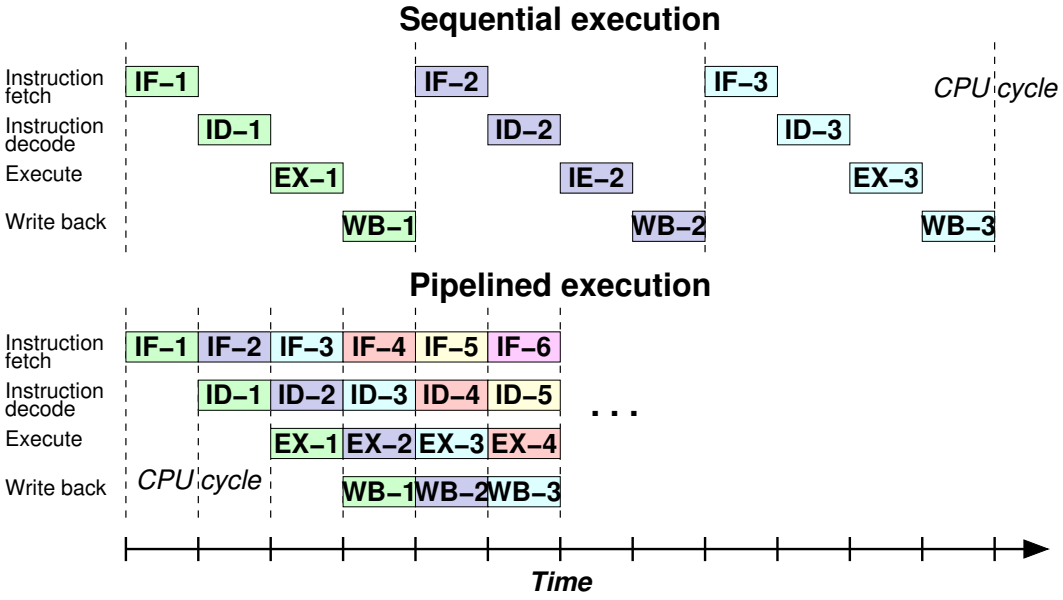


Figure 2.5: Sequential execution vs. Pipelined execution. From [Zuk09].

2.3.1. CPU

The central processing unit (CPU) is the part of a computer system that executes instructions of a computer program. It carries out each instruction of the program, to perform the basic arithmetical, logical, and input/output operations of the system. The CPU manipulates data, which can be stored in a number of places: registers, caches, RAM memory (from fastest to slowest). The instructions that it execute are also stored in memory or cache. CPU operates in *cycles* issued by a clock with a frequency in order of magnitude of around a few GHz in modern processors. In most of experiments we use cycles instead of time as a metric to measure performance. The elapsed cycles can be read using hardware counters.

2.3.2. Pipelined execution

Carrying out an instruction takes more than one CPU cycle. The execution consists of various stages:

- Instruction Fetch (IF) – get the next instruction to execute.
- Instruction Decode (ID) – decode the instruction from the binary operation code.
- Execute (EX) – perform the requested task. This can involve multiple different devices such as arithmetic logic units (ALU), floating-point units (FPU) or load-store units (SPU).
- Write-back (WB) – save the result.

These stages are executed by different components of the CPU. Execution of different stages of different instructions could therefore be overlapped. This way the CPU can accept a new instruction every cycle.

The first Pentium processors were able to execute two paired instruction simultaneously. From that point the pipeline depth was increasing until the Pentium 4 Prescott processors,

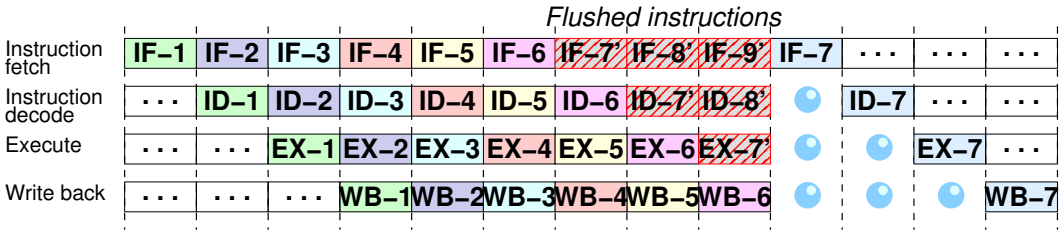


Figure 2.6: CPU bubble due to branch misprediction. From [Zuk09].

where it reached 31 stages. Due to problems with such deep pipelining described in Section 2.3.3, the pipelines got a little shorter in newer architectures. The Nehalem processors have 16 stages pipeline. A comparison of pipelined execution and sequential execution is shown in Figure 2.6.

2.3.3. Hazards and how to deal with them

The problem with deeply pipelined execution is that the instructions taken into the pipeline have to be independent. If the next instruction depends on the result of a previous one, CPU has to wait and cannot fully utilize the pipeline and performance is sub-optimal. This can be due to two types of situations called *hazards*: data hazards and control hazards.

A *data hazard* is when an instruction cannot be executed because it operates on an input that is not ready. For example in

```
c = a + b;
e = c + d;
```

the first addition has to finish before the second one can start.

A *control hazard* is a result of conditional jumps and branching of the code. To fill the pipeline the CPU needs to know what instructions will be executed next. If the code has conditional jumps ("if" statements) it might be impossible to schedule next instructions, causing stalls in the pipeline.

Out-of-order execution

To deal with the problem of instruction independence, a CPU can execute instructions out of order. Even though a program is a sequence of instructions, when reaching an instruction that cannot be put into execution due to a hazard, the CPU can look ahead and detect which of the upcoming instructions are independent and can be executed immediately. The reorder buffer in a Nehalem processor can contain 128 entries.

Branch prediction

To avoid waiting for the calculation of the outcome of a conditional jump, processors try to predict the outcome and speculate ahead, taking into execution instructions from one of the branches. The CPU bases its decision on a history of previous results of the conditional jump.

On one hand branch prediction helps performance if the prediction is correct, but if the branch is mispredicted the speculated instructions have to be undone, which imposes a penalty. In Nehalem processors this penalty is at least 17 cycles. If the pattern of whether the branch is taken or not is not regular, with a selectivity close to 50%, no sophisticated

branch prediction algorithm can help and performance is severely degraded. Therefore, such situations have to be avoided.

In our experiments we measure the number of mispredicted branches with hardware counters.

2.3.4. CPU caches

The memory system in a modern computer is hierarchical.

Registers can be called the fastest type of memory. There are 16 general purpose registers in a x86_64 CPU that are named and manipulated explicitly in an assembly program. In fact the CPU has more registers, which it can use to execute more instructions simultaneously in a pipeline. The access latency of a register is about 1-3 cycles.

Main memory is also accessed explicitly (see more in Section 2.3.6) and its capacity is in gigabytes, but the access latency is much higher, in order of hundreds of CPU cycles.

Because of that, there is a number of cache layers between the registers and main memory. These caches work implicitly and cannot be controlled by the programmer. The accessed data and instructions are automatically put into and evicted from the caches.

A Nehalem processors has three levels of cache. The L1 cache is split into an instruction cache and data cache, each with a capacity of 32 kB per CPU core. The latency for accessing L1 cache is 4 cycles, so it is almost as fast as registers. The L2 cache has 256 kB per CPU core and the access latency is 11 cycles. There is also a L3 cache with a capacity of 8 MB shared by the whole processor, but the access latency for that one goes up to 38 cycles. The data into the caches is loaded and stored in 64 byte cache-lines.

In VectorWise we try to achieve that the data vectors that we currently operate on fit and are permanently kept in the L2 cache.

2.3.5. SIMD instructions

We often encounter a situation where we execute the same operation on multiple data elements. This is especially the case in vectorized execution, where we perform basic operations on whole vectors of data in tight loops.

Processors are equipped with special instructions that can perform one operation on multiple data elements at once. This is called Single Instruction, Multiple Data (SIMD) model. It was started with MMX instruction set in the Pentium 1 processors, which were sharing registers with the floating point operations. More recent processors have a SSE instruction set, with its own set of registers. Nehalem CPUs support SSE version 4.2. SSE 4.2 defines 8 new 128-bit registers (`xmm0` to `xmm7`) and supports 54 instructions. Using it we can for example add and subtract 16 single-byte values, multiply 8 single-byte integers into 2-byte results etc. in single instruction.

Using SSE in vectorized operations is possible thanks to the column-wise orientation of vectors. Multiple values from one attribute are adjacent in memory, so that they can be moved into a SSE register with a single instruction. If the tuples were stored together row-wise, we would have to gather the values from different tuples from separate memory locations, which would make it an unprofitable endeavour.

2.3.6. Virtual memory and TLB cache

Unfortunately, not all database operations can be executed on data that fits in the processor caches. Sometimes we need to access random memory locations in very big data structures, for example while probing a hash table. Memory accesses to RAM memory are much more

expensive, but the performance penalty can be even bigger because of the way virtual memory is implemented.

A process memory is addressed in virtual addresses instead of physical ones. This way the process sees its memory as a contiguous and homogeneous address space, while in fact it can have multiple fragments of physical memory assigned by the operating system, or even have part of its memory swapped to disk.

Virtual memory consists of pages which are usually 4 kB in size (or 2 MB if big pages are enabled). A virtual memory address consists of the page address and an offset in the page. The page address has to be translated into an address in physical memory. This is done in hardware using a Translation Lookaside Buffer (TLB), which holds the translations for recently accessed tables.

This buffer's capacity is limited, so when we access random memory locations, we often hit virtual memory pages for which translations are not in the TLB, and we cause a TLB miss. To handle a TLB miss, the translation has to be read from a directory page. A directory page is itself a normal 4 kB virtual memory page and it can hold 512 page translation entries, covering a span of 2 MB of memory. If the directory page's translation is not in the cache either, we can hit another TLB miss while accessing it. Then we have to access a higher level directory page, which holds translations for 512 lower level directory pages, which corresponds to a span of 1 GB of memory. Accessing a random location in a huge span of memory can actually force us to perform multiple memory accesses for handling TLB misses, causing very high latency. We can either try to avoid such situations or provide other operations that can be out-of-order executed to hide that latency.

In recent architectures we observe a significant increase in TLB caches size. A Nehalem CPU has 2 levels of TLB and can hold 64 entries of 4 kB data pages and 64 entries of 4 kB instruction pages on the first level, and 512 entries of 4 kB pages on the unified second level.

The amount of TLB misses can be measured using hardware counters.

2.4. Related work on JIT query compilation

Just-In-Time (JIT) query compilation has been mostly considered so far as an alternative method for eliminating the ill effects of interpretation overhead. On receiving a query for the first time, the query processor compiles (a part of) the query into a routine that gets subsequently executed. We will generalise the current state-of-the-art JIT techniques using the term "single-loop" strategies, as these try to compile the core of the query into a single loop that iterates over tuples. This is in contrast with vectorized execution, which decomposes operators into multiple basic steps, and executes a separate loop for each of them ("multi-loop"). We will discuss some of the existing database systems employing JIT query compilation in Section 2.4.

It will be the goal of this thesis to generate on the fly new, specialized primitives for combinations of operations, reducing a tree (DAG) of interpreted expressions into a single compiled function. The proposed model is to perform many operations in a single-loop, operating on one tuple in one iteration, but leave it in the framework of the vectorized engine. The evaluation in Chapter 3 shows that such a mixed approach is better than compiling everything in single-loop compilation.

The following sections briefly describes other academic and commercial database management systems that use JIT query compilation.

2.4.1. Origins

System R originally generated assembly directly, but the the non-portability of that approach led to its abandonment [D. 81].

2.4.2. HIQUE

HIQUE is an academic system developed at the University of Edinburgh and described in [KVC10, Kri10]. It coins the term *holistic* compilation. It generates and compiles code in C for whole queries at once, using an external call to the gcc compiler and dynamic linking. Compilation time of a whole TPC-H query is in the order of hundreds of milliseconds.

At the simplest level, all primitive operations such as comparisons, casts, data fetching etc. are generated and inlined, tailored for the specific data types and tuple layout. HIQUE code generation strives to achieve good cache locality and linear memory access patterns. It employs elements of block oriented processing by using units of work fitting in the L2 cache, which the authors call *pages*. The prototype system is limited to queries that can be executed with and Aggregation on top of Joins on top of a Selection over a Scan. All Selections have to be conjunctive and without negation, so they can be pushed down below the joins. The first group of generated functions perform data input and selections. Storage is NSM model, but unnecessary attributes are dropped during scan, to produce thinner tuples. HIQUE immediately calculates the selections during scan. Joins are hash-partitioned into L2 cache-fitting partitions on both sides of the join. Whole tuples, not references, are being copied into the partitions so that further memory accesses are local. Then either a nested loop join is used, or cache-fitting partitions are sorted and a merge join is performed, to achieve linear memory access pattern. JIT code generation allows HIQUE to generate specialized code for multi-way joins on a common key, which are common in analytical workloads. Such multi-way joins implemented as *join teams* can be very efficient [GBC98]. In aggregation HIQUE also prefers ordered aggregation, hash partitioned into L2 cache fitting chunks. With dynamically generated code many aggregate functions can be calculated at once.

In general, HIQUE performs a lot of in-memory intermediate materialization and data staging to use algorithms with linear memory access patterns and work on L2 cache fitting chunks. This can be seen as elements of vectorization.

The authors of HIQUE observe that the development process of implementing C code generation is relatively easy. Errors in code generation can be easily spotted by analyzing the generated C code. This way, the number of develop and test iterations required until code generation for a new algorithm is fully supported is reduced.

HIQUE claims TPC-H performance comparable to MonetDB/X100 system in [KVC10], not based on a direct benchmark but by comparing its execution times with times given in [ZNB08].

2.4.3. Hyper

The recently announced Hyper system is developed at Technical University of Munich [KN10].

Contrary to HIQUE, the algorithms Hyper use are intended to avoid any intermediate materialization as much as possible. The generated code breaks operator boundaries, combining as much code together in a single loop and keeping the intermediate results in registers. Materialization is done only when strictly necessary, when a *pipeline breaker* is encountered, e.g. if one side of the join has to be put into a hash table or an aggregation result has to be calculated. Each generated code fragment performs all actions that can be done within one part of the execution pipeline before materializing the result into the next pipeline breaker.

Another difference is that query execution follows a "push" model instead of "pull", with data being pushed towards the root of the query execution tree, instead of being pulled from the leaves.

What makes Hype different from other systems is that it does not generate code in a high level language but in LLVM assembly-level intermediate representation (IR) language. LLVM is a collection of modular and reusable compiler-construction tools that will be described in Section 4.1. The LLVM IR is architecture-independent, but it uses assembly-level instructions operating on virtual registers. Code generation may be harder than in higher-level language, but provides more precise control over the result. The LLVM code optimization library is used to optimize the assembly code. It provides optimization passes just as in different compilers, but using LLVM directly Hyper can control what passes and in what order are performed or add its own custom optimization passes, adapted to the specific code it generates.

2.4.4. JAMDB

The JAMBD system comes from IBM Almaden Research Center [RPML06]. The major characteristic that distinguishes it from others is that it is implemented in Java and uses reflection and JIT present in Java Virtual Machines.

The system has no real storage layer, all data is Java objects in memory. It does not deal with the problems of ensuring ACID properties. A database table schema corresponds simply to a Java class and one object is a row of the table. Foreign keys are just references to other Java objects in memory and indexes are Java HashMaps. Joins are supported only on these explicitly created join indexes. JAMDB depend on Java Virtual Machine for buffers and memory management.

Code generation is therefore relatively simple. Almost nothing is needed for joins, as they are just reference following or Java HashMap lookups. For selections JAMDB reorders the predicates based on predicted selectivity, to get the most filtering condition first. Nothing is however said about reordering them dynamically at runtime if the observed selectivity differs from the expected one. Most of the code generation is actually just inlining correct comparison, casting and fetching methods for different datatypes.

Their main claim is that using Java Virtual Machine and generating Java classes lets them take advantage of JVM's built in JIT compilation capabilities. They generate new Java classes code and load them directly using Java's class loader. When compiling operations together JAMDB avoids virtual function calls and reflection in Java. The authors claim that the JVM JIT compiler will analyze and optimize the executed code in runtime, which will result in performance comparable to a hard coded C system.

The provided system evaluation is very limited. JAMDB is compared only against DB2, achieving speedup not more than 3x over an interpreted system.

2.4.5. Other

AT&T developed a system called Daytona. In [Gre99] not much is written about how and what code is generated and about the overall infrastructure. It rather describes the syntax and capabilities of a custom query language called Cymbal. In the Daytona system there is no database server. Each query template is compiled into a standalone program, that accesses the data directly. A query template is a query with some parameters placeholders, and the generated program is called with these parameters filled to perform the actual query. The database itself is stored as disc files directly accessed by the query programs. All concurrency control, scheduling, locking etc. is left to the operating system file locking and

scheduling mechanisms. The system does not deal with ACID properties itself. No discussion or measurements are provided about Daytona's performance.

Another system known to use compilation is the ParAccel [Par10] commercial analytical database system. No details on its inner working are publicly released however.

2.5. Summary

In this chapter we introduced all concepts necessary for understanding the rest of this thesis.

A description of database management system construction in general, and VectorWise DBMS in particular, have been provided. In addition to VectorWise, which is a representative of a vectorized system, we analyzed and described existing systems that implement JIT compilation.

The reasoning behind many of the algorithms and methods used in this thesis lies in the architecture of modern hardware, so an introduction to some hardware concepts was also needed.

Chapter 3

Vectorization vs. Compilation

Vectorization and JIT compilation have been proposed as two different techniques for reducing the interpretation overhead of a database management system. In this chapter we try to shed light on the question of how state-of-the-art compilation strategies relate to vectorized execution for analytical database workloads on modern CPUs and is it worth combining them. For this purpose, we carefully investigate the behavior of vectorized and compiled strategies in three use cases: Project, Select and Hash Join. One of the findings is that compilation should always be combined with block-wise query execution. Another contribution is identifying three cases where "loop-compilation" strategies are inferior to vectorized execution. As such, a careful merging of these two strategies is proposed for optimal performance: either by incorporating vectorized execution principles into compiled query plans or using query compilation to create building blocks for vectorized processing. [1]

3.1. Experiments overview

We have chosen three case studies which show how to exploit the capabilities of modern processors by combining vectorization with loop-compilation. These case studies cover operators important for database performance, such as Project, Select and hash-based operators (using the example of HashJoin). Improving their performance would result in significant speedup of database engine as a whole.

As our first case, Section 3.2 shows how in Project expression calculations loop-compilation tends to provide the best results, but that this hinges on using block-oriented processing and SIMD. Thus, compiling expressions in a tuple-at-a-time engine may improve some performance, but falls short of the gains that are possible. This case study shows the problems with effectively using SIMD capabilities of modern processors.

In Section 3.3, our second case is Select, where we show that branch mispredictions hurt loop-compilation when evaluating conjunctive predicates. In contrast, the vectorized approach avoids this problem as it can transform control-dependencies into data-dependencies for evaluating booleans (along [Ros02]). In Section 3.4 we consider a Select with disjunctive predicates, which is a little more complicated for the vectorized approach, but it still outperforms loop-compilation. These case studies show the problems of maintaining deeply pipelined execution and avoiding branches. Section 3.5 gives a synthesis of the Project and Select cases.

[1] Part of material from this chapter was published as a paper at the DaMoN 2011 workshop in Athens [SZB11].

The third case, described in Section 3.6, concerns building and probing large hash-tables, using a HashJoin as an example. Here, loop-compilation gets CPU cache miss stalled while processing linked lists (i.e. hash bucket chains). We show that a mixed approach that uses vectorization for chain-following is fastest and robust to the various parameters of the key lookup problem. This case study shows the importance of optimal memory access patterns and exploiting the processor's ability of out-of-order execution. Since hash tables are used not only in HashJoin but in different operators, it can be applied also to e.g. Aggregation.

In Section 3.7 we discuss aspects of vectorization and computation other than performance – software engineering, system complexity and maintainability.

These findings lead to a set of conclusions which we present in Section 3.8.

3.1.1. Experimental setup

We used the PAPI library [TJYD09] to collect the actual number of clock cycles, retired instructions, branch mispredictions, and TLB misses in our microbenchmarks. Between 100 and 1000 repetitions of each measurement has been performed, to make sure that the gathered results are stable and smooth. We used a 2.67GHz Nehalem core, on a 64-bits Linux machine with 12GB of RAM. The gcc 4.4.2 compiler was used for most of the experiments. We explicitly mention places where we also used icc 11.0 or clang with LLVM 2.9.

3.2. Case study: Project

Inspired by the expressions in Q1 of TPC-H we focus on the following simple Scan-Project query as micro-benchmark:

```
SELECT l_extprice*(1-l_discount)*(1+l_tax) FROM lineitem
```

The scanned columns are all decimals with precision two. VectorWise represents these internally as integers, using the value multiplied by 100 (decimal with precision 2). After scanning and decompression it chooses the smallest integer type that, given the actual value domain, can represent the numbers. The same happens for calculation expressions, where the destination type is chosen to be the minimal-width integer type, such that overflow is prevented. In the TPC-H case, the price column is a 4-byte integer and the other two are single-byte columns. The addition and subtraction produce again single bytes, their multiplication a 2-byte integer. The final multiplication multiplies a 4-byte with a 2-byte integer, creating a 4-byte result.

We have run the query using different vector lengths, from 1 to $4M$ tuples. The results presented in Figure 2 present the cost of calculating the whole expression per one tuple.

3.2.1. Vectorized Execution

The expression is calculated using map primitives shown in the upper part of Algorithm 2. In section 2.2 it was discussed that primitives are generated for combinations of operations and types. A question is: how many such combinations should be considered?

Currently, VectorWise pre-generates map primitives where the input vectors and the results all have the same type. Focusing on integer types (1-, 2-, 4-, 8- byte; signed/unsigned) there are 8 combinations of functions to generate for each operation (which is further multiplied by the `_col` and `_val` combinations). However, such functions do not suit all needs of the benchmarked expression: at one point we want to multiply 1-byte integers into a 2-byte

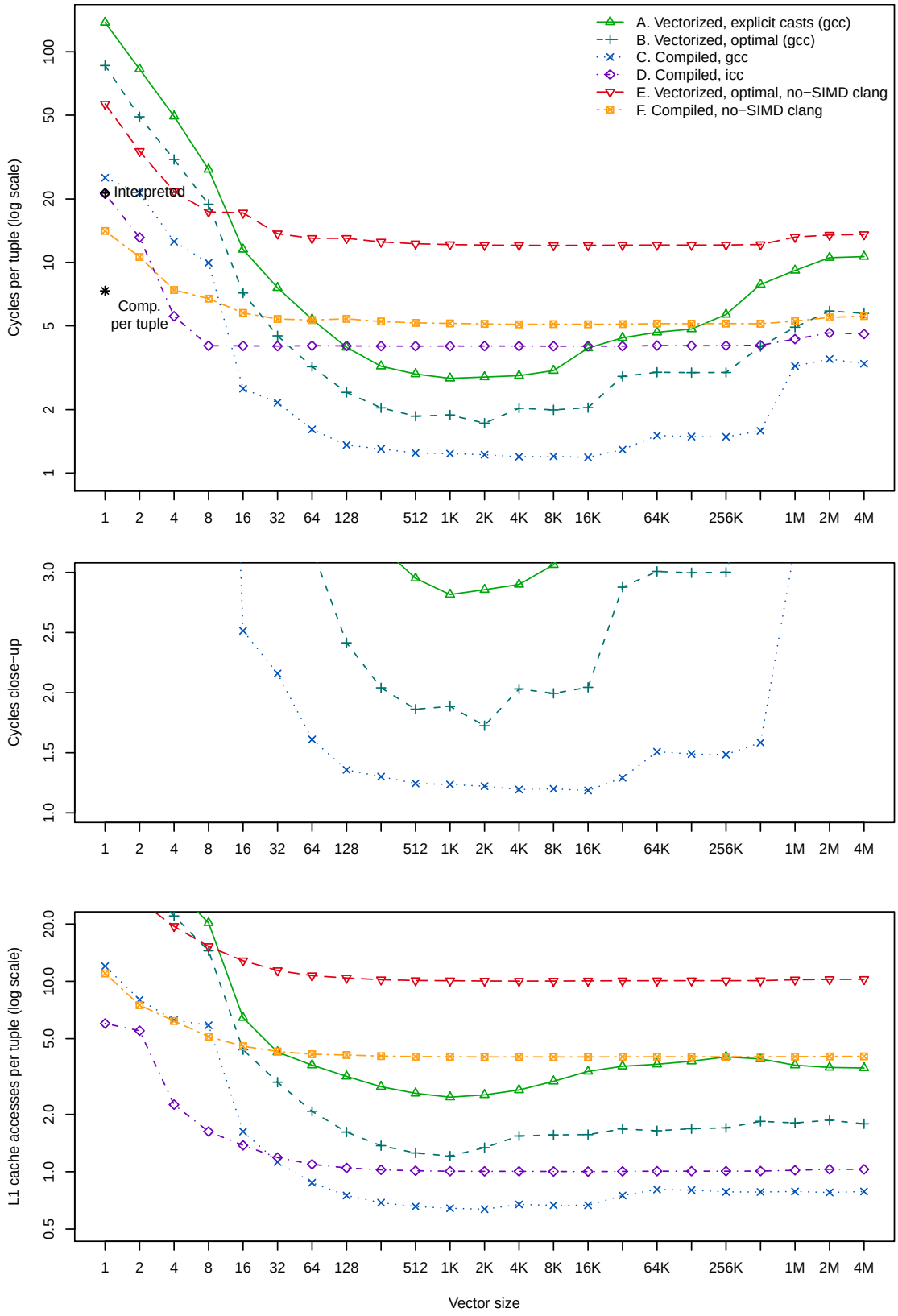


Figure 3.1: Project micro-benchmark: vectorized and compiled implementations of an arithmetic expression. Different compilers are used to show differences in SIMDization capabilities. Two vectorized implementations: optimal and with forced type coercions show the trade-off between performance and the amount of necessary pregenerated code.

Algorithm 2 Implementation of an example query using vectorized and compiled modes. Dynamically compiled primitives, such as map_compiled(), follow the same template as pregenerated vectorized primitives, but may take arbitrarily complex expressions as OP. Handling of selection vector is omitted for compactness.

```
const uidx VECLENGTH=1024;
// input
uint l_extprice[VECLENGTH]; uchr l_discount[VECLENGTH],l_tax[VECLENGTH];
uchr v1 = 100, v2 = 100;
// output
uint res[VECLENGTH];
// intermediates
uchr tmpuchr1[VECLENGTH],tmpuchr2[VECLENGTH];
usht tmpusht1[VECLENGTH],tmpusht2[VECLENGTH],tmpusht3[VECLENGTH];
uint tmpuint1[VECLENGTH];

// Vectorized code with need for explicit casting:
map_uchr_add_uchr_col_uchr_val(VECLENGTH, tmpuchr1, uchr1, v1, sel);
map_uchr_sub_uchr_val_uchr_col(VECLENGTH, tmpuchr2, v2, uchr2, sel);
map_uchr2usht_uchr_col(VECLENGTH, tmpusht1, tmpuchr1, sel);
map_uchr2usht_uchr_col(VECLENGTH, tmpusht2, tmpuchr2, sel);
map_usht_mul_usht_col_usht_col(VECLENGTH, tmpusht3, tmpusht1, tmpusht2, sel);
map_usht2uint_usht_col(VECLENGTH, tmpuint1, tmpusht3, sel);
map_uint_mul_uint_col_uint_col(VECLENGTH, res, uint1, tmpuint1, sel);

// Optimal vectorized code with primitives for more combinations of types:
map_uchr_add_uchr_val_uchr_col(VECLENGTH, tmpuchr1, &v1, l_discount, sel);
map_uchr_sub_uchr_val_uchr_col(VECLENGTH, tmpuchr2, &v2, l_tax, sel);
map_usht_mul_uchr_col_uchr_col(VECLENGTH, tmpusht1, tmpuchr1, tmpuchr2, sel);
map_uint_mul_uint_col_usht_col(VECLENGTH, res, l_extprice, tmpusht1, sel);

// Compiled equivalent of this expression:
uidx map_compiled(uidx n, uint *res, uint *col1, uchr* col2, uchr* col3 /*, uidx* sel */) {
  for(uidx i=0; i<n; i++)
    res[i]=col1[i]*((100-col2[i])*(100+col3[i]));
}
```

result, and then multiply 4- and 2- byte integers together. We need additional primitives, which cast a vector into a vector of another type, to coerce it to be suitable to be used with a proper arithmetic map primitive. This not only constitutes a redundant additional operation, but also uses more memory, as an additional intermediate vector has to be used to save the result of casting. This may make the operation working space memory needs grow beyond CPU cache, causing performance degradation.

If the map primitives were able to accept parameters of different types, casting would be zero-cost, performed during moving the operands from memory to register, and less intermediate values would be materialized. The performance can be compared looking at the "explicit casts" and "optimal" lines in Figure 3.1.

However, such flexible behaviour comes at a cost of having to pregenerate an order of magnitude more combinations of primitives. Consider a binary operation on integer types. Originally there were 8 variants generated because of types combinations. Now we would like the operation to be able to take parameters also from shorter types and also convert between signed and signed.

- An operation producing a signed/unsigned 1-byte result will have to accept two parameters, of which each could be signed/unsigned 1-byte – 2^2 combinations for two parameters.

- An operation producing a signed/unsigned 2-byte result will need support for signed/un-

signed 1- and 2- byte parameters – 4^2 combinations.

- Similar for 4 and 8 byte datatypes.

As a result, instead of 8 combinations there are

$$2 \cdot 2^2 + 2 \cdot 4^2 + 2 \cdot 6^2 + 2 \cdot 8^2 = 240$$

This is a 30-fold increase in size of code that needs to be generated for binary operations. The results in Figure 3.1 show that this approach can lead to significant performance gains, but such an increase in executable size might be unacceptable. It can alone, if nothing else, be a reason for using JIT compilation to generate simple primitives with the required combination of types.

3.2.2. Loop-compiled Execution

The lower part of Algorithm 2 shows the compiled code that a modified version of VectorWise can now generate on-the-fly: it combines vectorization with compilation. Such a combination in itself is not new ("compound primitives" [BZN05]), and the end result is similar to what a holistic query compiler like HIQUE [KVC10] generates for this Scan-Project plan, though it would also add Scan code. However, if we assume a HIQUE with a simple main-memory storage system and take `l_tax`, etc. to be pointers into a column-wise stored table, then `map_compiled()` would be the exact product of a "loop-compilation" strategy.

The most obvious reason for using JIT compilation in the context of compound operations in a projection, is to avoid materialization of intermediate results and the necessity of type coercions.

3.2.3. SIMDization using SSE

Simple independent arithmetic operations on columns of data, such as performed in map-primitives, are easy to SIMDize in the vectorized implementation. With a *single* SSE instruction, modern CPUs can add and subtract 16 single-byte values, multiply 8 single-byte integers into 2-byte results, or multiply two 4-byte integers into 8-byte results (cast back into 4-byte type) [sse]. It is reasonable to expect that a compiler that claims support for SIMD should be able to vectorize the trivial loop in the `map_` functions.

Thus, 16 tuples in the analyzed expression could be processed by the vectorized implementation with 12 calculations:

- 1 for single-byte addition,
- 1 for single-byte subtraction,
- 2 for multiplications of single-bytes into a 2-byte results,
- 8 for multiplication of 2-byte and 4-byte into a 4-byte results.

The input for 16 tuples are $16 \cdot (1 + 1 + 4) = 96$ bytes of memory. Intermediate results of vectorized execution are $16 \cdot (1 + 1 + 2) = 64$ bytes of memory. The final result is $16 \cdot 4 = 64$ bytes of memory. We therefore need to load $96 + 64 = 160$ bytes and store $64 + 64 = 128$ bytes of memory for every 16 tuples. Using SSE we will be loading 16 bytes into an SSE register in one memory operation, so we need 18 operations.

The vectorized implementation with explicit casting needs to cast 1-byte results of `100 + l_tax` and `100 - l_discount` into 2-byte type, and cast the 2-byte results of `(100+l_tax)`

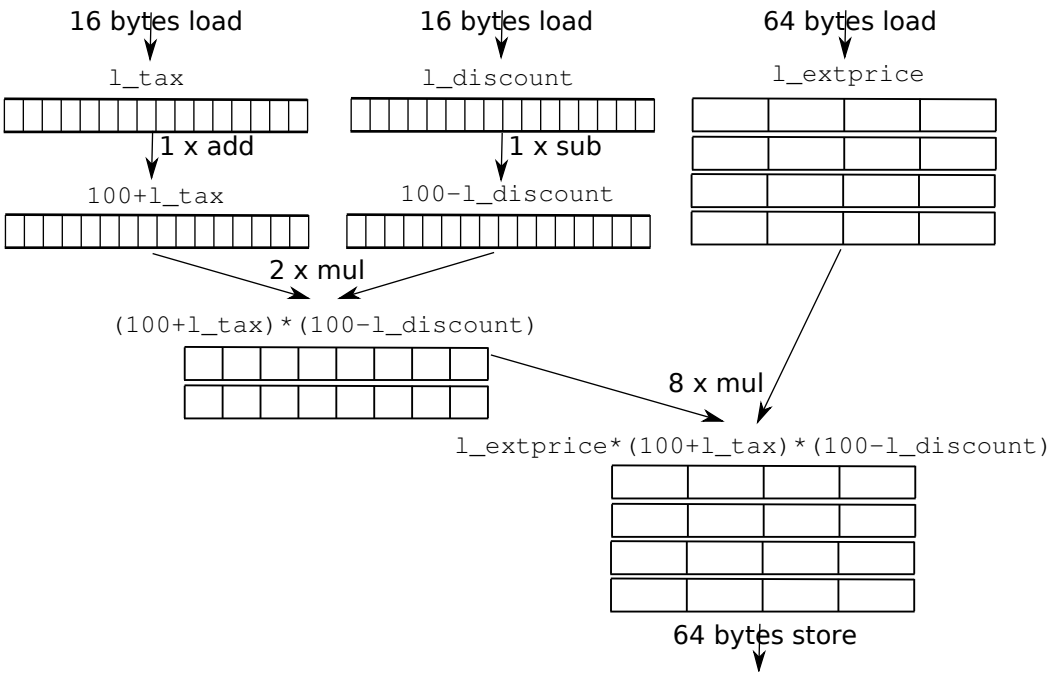


Figure 3.2: Calculation of `l_extendedprice*(100+l_tax)*(100-l_discount)` with SSE. The diagram shows number of operations that have to be performed per 16 tuples. In the "loop-compiled" implementation only the input attributes and final result have to be loaded from/to memory. A vectorized implementation additionally has to store all intermediate results in memory.

`* (100-l_discount)` into 4-byte type. For the additional intermediate results we need $16 + 16 + 16 \cdot 2 = 64$ bytes per 16 tuples that have to be loaded and stored. The total memory traffic is therefore 224 bytes loaded and 192 bytes stored, which is 22 memory operations.

The loop in ,,loop-compiled" implementation is more complicated to vectorize in an optimum way, because different operations have to be vectorized with different block sizes. The loop has to be decomposed to process one 16-tuple single-byte addition and subtraction, then two 8-tuple 2-byte multiplications and finally eight 4-byte multiplications, storing intermediate results in SSE registers. This requires more padding and border-case handling, but dramatically reduces the number of memory operations. We only need to load the input parameters and store the final result. This is 96 bytes to load and 64 bytes, which can be done in 10 memory operations for 16 tuples.

Without SIMD each input attribute, intermediate result and final result has to be loaded and stored separately for each tuple. This requires $3+1$ stores (3 intermediate results, 1 final result) and $3 + 3$ loads (3 input attributes, 3 intermediate results), for a total of 10 memory operations per tuple. Compiled implementation without SIMD only needs $1+3 = 4$ memory operations per tuple, because it does not produce intermediate results.

This reasoning about the number of memory and calculation operations is presented on a diagram in Figure 3.2. The bottom graph in Figure 3.1 shows number of L1 cache accesses per tuple made by different implementation which is roughly in line with this discussion.

It is a question whether the compiler will also be able to automatically vectorize the ,,loop-compiled" compound operation. We performed our experiments on three compilers

- `gcc 4.4.2` with options `-O3 -msse4 -ftree-vectorize`

- `icc 11.0` with options `-O3 -xSSE4.2`
- `clang/llvm 2.8` with options `-O3`

Currently, LLVM and `clang` does not perform auto-SIMDization using SSE. However there is a project to add this functionality underway [GZA+11]. Hopefully, when it is finished it will work as well as in `gcc`. For now, `clang` run shows the performance of non-SIMDized code.

Only `gcc` line is shown in Figure 3.1 for the vectorized implementations (lines "A" and "B"), because there was no difference between `gcc` and `icc` in this case. However, in the course of those experiments it turned out that in the general case `gcc` sometimes refuses to vectorize the trivial loops in `map_` primitives. It is some compiler sub-optimality, as tweaking little things in the code, like changing the loop iterator variable from unsigned to signed type, or using temporary local variables to store intermediates instead of inlining operations helped `gcc` to improve auto-vectorization. It was made sure that the `gcc` code used for the results was SIMDized by the compiler properly, but it required some manual tweaking. `icc` was indifferent to those changes and was able to auto-vectorize the `map_` primitives in all cases. Those results warn that SIMDization of the simple vectorized primitives also has its pitfalls. However, as this code is compiled statically, it can be inspected to make sure that it works as expected before releasing it. It is harder to make such guarantees for dynamically generated and compiled code.

`gcc` (line "C") performs much better than `icc` (line "D") in the ,,loop-compiled" implementation. Looking at the assembly revealed that `gcc` was able to vectorize the more complex loop in an optimal way described above, while `icc` chooses in its SIMD generation to align all calculations to the widest datatype (here: 4-byte integers). Hence, the opportunities for 1-byte and 2-byte SIMD operations working on more tuples at once are lost. Together with the need of additional padding and zero-extending the shorter data types, this makes `icc` implementation only slightly faster than the one not using any SIMD instructions.

If the operations are compiled together in an optimal way, the performance gain might be even 50% over the best possible vectorized version, and almost by a factor 3 over the implementation with explicit casting and coercions requiring less combinations of statically pre-compiled vectorized primitives. This however depends on the compiler being able to SIMDize the operation properly. Worse yet, such a sub-optimality might be much harder to notice. As the code is generated in run-time, it cannot all be inspected. Sub-optimal behaviour for some combinations of operations might easily go unnoticed.

Until the LLVM auto-SIMDization project is finished and introducing SSE works as well in `clang` as it's in `gcc`, loop-compilation of expressions working on full vectors of data using `clang` (line "E") as the JIT compilation will work worse than vectorized code using SSE. The vectorized code compiled with `clang` without SSE (line "F") works over ten times slower than the `gcc` one with SSE (line "B"). Care has to be taken how good which compiler handles auto-vectorization, but `gcc` results in loop compilation are very promising.

3.2.4. Tuple-at-a-time compilation

Much of the performance of map operations has to be attributed to the ability to use SSE SIMD operations on many tuples at a time. This however is possible only thanks to the overall vectorized architecture of the DBMS. An engine like MySQL, whose whole iterator interface is tuple-at-a-time, cannot exploit it, as it has just one tuple to operate on at any moment. We have therefore also measured the possible gains of compiling map operations in

a DBMS working tuple-at-a-time, to see what the potential gains from compilation would be in such a system.

The black star and diamond in Figure 3.1, correspond to situations where primitive functions work tuple-at-a-time. The non-compiled strategy is called "interpreted", here. Tuple-at-a-time primitives are conceptually very close to the functions in Algorithm 2 with vector-size `n=1`, but lack the for-loop. We implemented them separately for fairness, because these for-loops introduce loop-overhead. This experiment shows that if one would contemplate introducing expression compilation in an engine like MySQL, the gain in performance for expression calculation could be something like a factor 3 (23 vs 7 cycle/tuple). The absolute performance is still much worse from what block-wise query processing offers (7 vs 1.2 cycles/tuple), mainly due to missed SIMD opportunities, but also because the virtual method call for every tuple inhibits speculative execution across tuples in the CPU. Even worse, in tuple-at-a-time query evaluation function primitives in OLAP queries only make up a small percentage ($< 5\%$) of overall time [BZN05], because most effort goes into the tuple-at-a-time operator APIs. When considering expression compilation, the overall gain without changing the tuple-at-a-time nature of a query engine can therefore at most be a few percent, making such an endeavour questionable.

3.2.5. Effect of vector length

The general trend of decreasing interpretation overhead with increasing vector-size until around one thousand and performance deteriorating due to cache misses if vectors start exceeding the L1 and L2 caches has been described in detail in [Zuk09, BZN05].

A final observation is that compiled map-primitives are less sensitive to cache size, since they use less intermediate results, so that a hybrid vectorized/compiled engine can use larger vector-sizes, because larger but fewer vectors will still fit in cache.

3.2.6. Project under Select and compute-all optimization

Map operations in Project were very fast, because they were heavily SIMDized when working on full vectors of data. Suppose now, that the calculated expression is put under a selection condition, as in the following query:

```
SELECT l_extprice*(1-l_discount)*(1+l_tax) FROM lineitem
WHERE ...
```

We will deal with selections in Sections 3.3 and 3.4 and then provide a synthesis in Section 3.5. For now we need to know that VectorWise handles selections with a vector of indexes of selected tuples that is passed on to the map operations, which in turn will perform operations only on the selected elements instead of full vectors.

Working with a selection vector slows execution down, because

1. less importantly, there are more memory accesses, as the elements of the vectors are not accessed directly, but through indexes stored in selection vector.

2. more importantly, SSE SIMD instructions cannot be used, as the operations are not performed on contiguous fragments of memory.

Because of that, ignoring the selection vector and performing computation on all tuples might be faster, and it is safe for most operations (when the operations do not have any unwanted side effects, e.g. like divisions, where possibility of division by zero error has to be

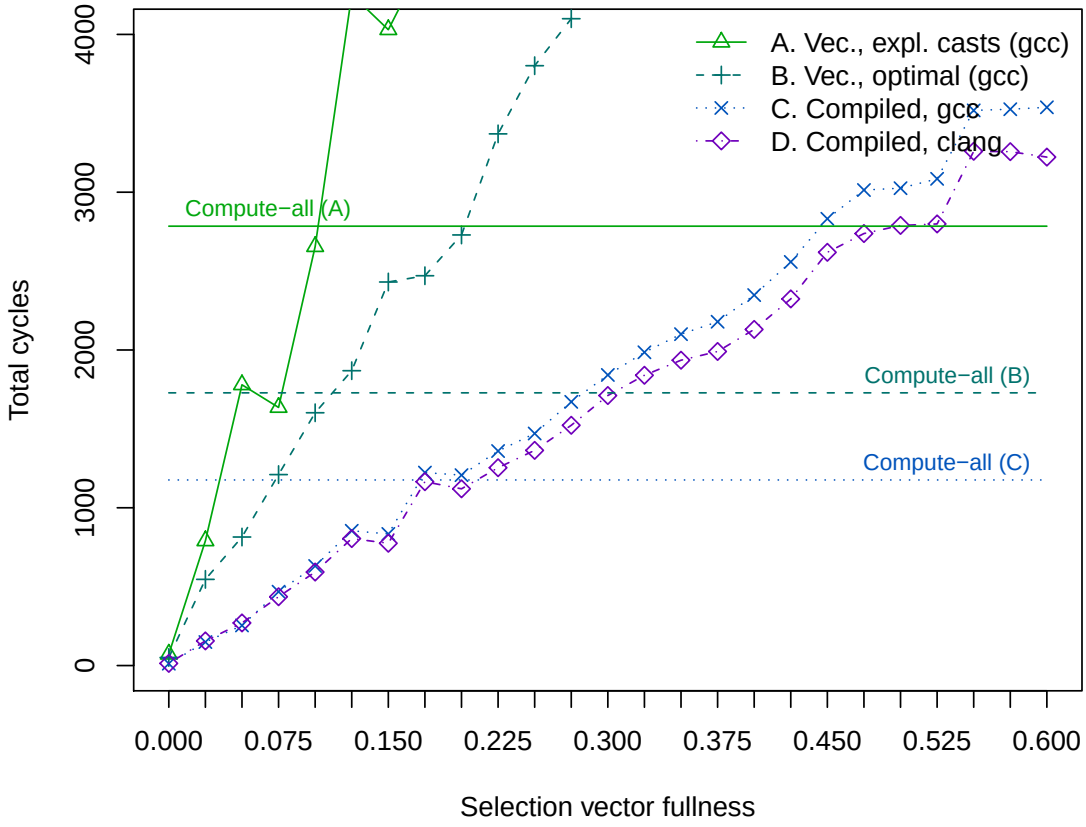


Figure 3.3: Project under Select. If the number of tuples in the selection vector exceeds a certain threshold, it is more profitable to make the computation on all tuples. The length of a full vector is 1024.

taken into account). The threshold selection vector fullness can be estimated with a simple heuristics. If there are V elements in a full vector and N are selected and the operation can be SIMDized to work on k tuples in one instruction, then if $\frac{N}{V} > \frac{1}{k}$, it should be faster to drop the selection vector for the reason of SSE alone. This optimization can be decided at run time for each vector of data independently and is called "compute-all".

Results in Figure 3.3 show, that it is beneficial to perform compute-all on vectorized implementations (line "A" with explicit casts, and line "B" without) for as few as around 10% selected tuples. An important aspect to consider when using `clang` for JIT compilation of compound primitives is that since the current (2.9) version does not support auto-SIMDization, the biggest advantage of compute-all is broken. Because of that, we don't show a "compute-all" line for the `clang` compiled implementation (line "D"). The vectorized implementation using compute-all will outperform the `clang` compiled one when the fraction of selected tuples is high enough. The `gcc` compiled implementation using compute-all (line "C") shows how the JIT compiled primitives work work when compiled with auto-SIMDization.

Even without SSE, the compiled implementation is faster than the vectorized one that would currently be used in VectorWise when the selection vectors are less than half full. It is profitable to use JIT compilation for compound map calculations under a selection vector in such cases.

Algorithm 3 Implementations of conjunctive selection. The vectorized implementation evaluates the three conditions lazily. Each of the predicates produces a selection vector, with which the next predicate is evaluated. The resulting selection vector is overwritten at each step with a new, shorter one. In the end the selection vector contains only positions for which all three conditions evaluated true. For mid selectivities, branching instructions lead to branch mispredictions. Either `__branching` or `__datadep` versions of selection primitives are dynamically selected by VectorWise, depending on the observed selectivity. Branching cannot be avoided in loop-compilation, which combines selection with other operations, without executing these operations eagerly. There are four alternative implementations of the loop body, with different trade-offs between branching and eager calculation.

```
/* Vectorized implementation. */
const uidx LENGTH = 1024;
// Input
uint col1[LENGTH], col2[LENGTH], col3[LENGTH], v1, v2, v3;
// Resulting selection vector.
uidx res[LENGTH];
uidx n1 = sel_lt_uint_col_uint_val(LENGTH, res, col1, &v1, NULL);
uidx n2 = sel_lt_uint_col_uint_val(n1, res, col2, &v2, res);
uidx retn = sel_lt_uint_col_uint_val(n2, res, col3, &v3, res);

/* Compiled version: four implementations of the above. */
// (C.) all predicates branching ("lazy")
uidx c0001(uidx n,uint* res,uint* col1,uint* col2,uint* col3,uint* v1,uint* v2,uint* v3){
  uidx j = 0;
  for(uidx i=0; i<n; i++)
    if (col1[i]<*v1 && col2[i]<*v2 && col3[i]<*v3)
      res[j++] = i;
  return j; // return number of selected items.
}
// (D.) branching 1,2, non-br. 3
uidx c0002(uidx n,uint* res,uint* col1,uint* col2,uint* col3,uint* v1,uint* v2,uint* v3){
  uidx j=0;
  for(uidx i=0; i<n; i++)
    if (col1[i]<*v1 && col2[i] < *v2) {
      res[j] = i; j += col3[i] < *v3;
    }
  return j;
}
// (E.) branching 1, non-br. 2,3
uidx c0003(uidx n,uint* res,uint* col1,uint* col2,uint* col3,uint* v1,uint* v2,uint* v3){
  uidx j=0;
  for(uidx i=0; i<n; i++)
    if (col1[i]<v1) {
      res[j] = i; j += col2[i]<*v2 & col3[i]<*v3
    }
  return j;
}
// (F.) non-branching 1,2,3, ("compute-all")
uidx c0004(uidx n,uint* res,uint* col1,uint* col2,uint* col3,uint* v1,uint* v2,uint* v3){
  uidx j=0;
  for(uidx i=0; i<n; i++) {
    res[j] = i;
    j += (col1[i]<*v1 & col2[i]<*v2 & col3[i]<*v3)
  }
  return j;
}
```

3.3. Case study: Conjunctive select

We now turn our attention to a micro-benchmark that tests conjunctive selections:

```
WHERE col1 < v1 AND col2 < v2 AND col3 < v3
```

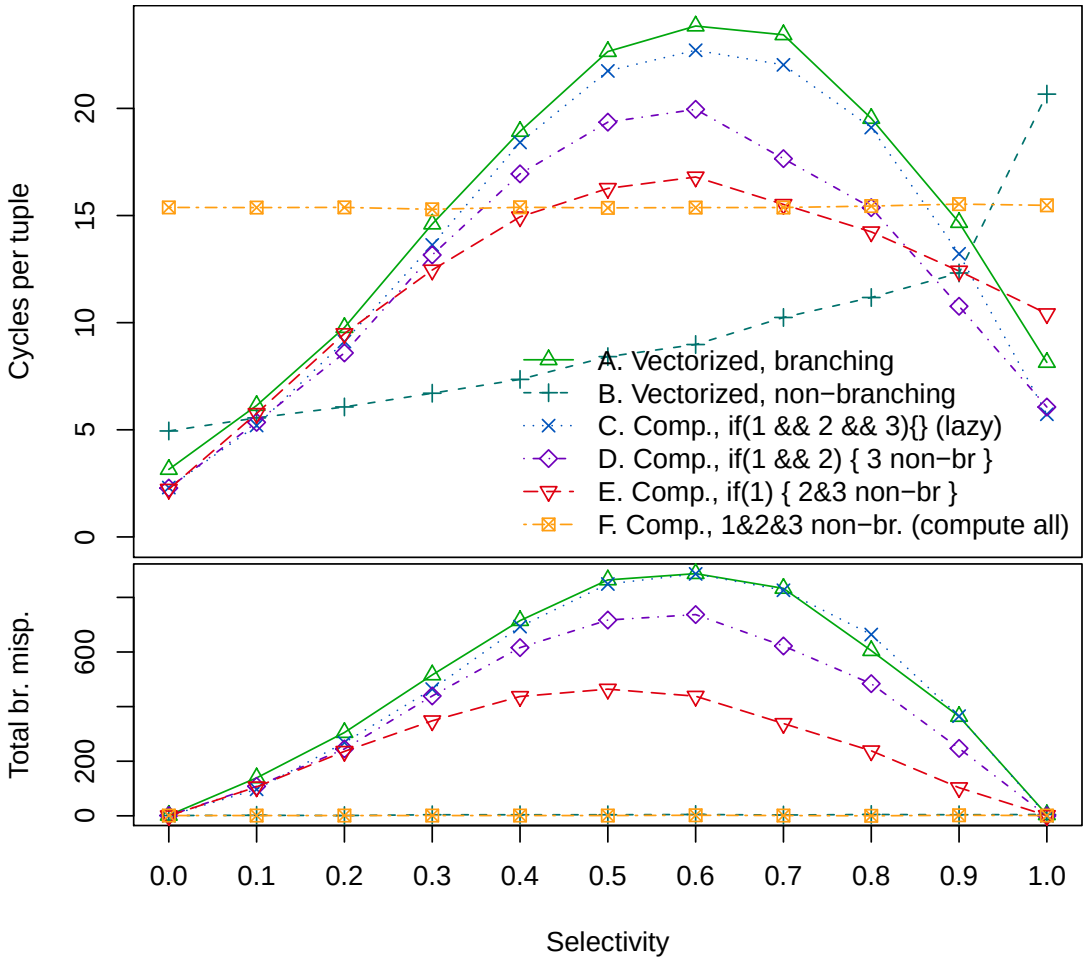


Figure 3.4: Conjunction of selection conditions: total cycles and branch mispredictions vs. selectivity

In this experiment each of the columns `col1`, `col2`, `col3` is an integer column, and the values `v1`, `v2` and `v3` are constants, adjusted to control the selectivity of each condition. The selectivity x on the x-axis in Figure 3.4 is the approximate fraction of input values for which each of the three conditions is true. Therefore, the overall selectivity of the selection is x^3. We performed the experiment on vectors of 1K input tuples.

3.3.1. Vectorized execution

With the way VectorWise uses selection vectors to handle selections it is very easy to evaluate a conjunctive condition in a lazy way. A primitive for first predicate outputs a selection vector with positions for which it evaluated to true. Then the second primitive can be called with this selection vector, to evaluate lazily only on those positions, which passed the first condition, and so on. As the resulting selection vector will get shorter with each step, one vector can be overwritten in each subsequent operation. No intermediate vectors are needed.

It was already mentioned in Algorithm 1 in Chapter 2 that VectorWise uses two alternative implementations of a selection primitive. The naive "branching" implementation evaluates the condition and saves the position into a selection vector only if the predicate is true. If

the selectivity of conditions is neither very low or high CPU branch predictors are unable to correctly guess the branch outcome. This prevents the CPU from filling its pipelines with useful future code and hinders performance. In [Ros02] it was shown that a branch (control-dependency) in the selection code can be transformed into a data dependency for better performance. In that implementation the position is always saved into a resulting selection vector, but the counter of selected positions is incremented by the outcome of the calculated condition – so the counter moves to the next position only if the condition evaluated to true. This approach needs to execute more instructions, but there are no branches, so for medium selectivity the pipeline does not get stalled in case of branch mispredictions. The VectorWise implementation of conjunctive selections uses a mechanism that chooses either the branching or non-branching strategy depending on the observed selectivity. It even re-orders dynamically the conjunctive predicates such that the most selective is evaluated first. As such, its performance achieves the minimum of the vectorized branching and non-branching lines in Figure 3.4.

3.3.2. Loop-compiled execution

Compiled implementations of conjunctive selection are shown in Algorithm 3. The gist of the problem is that the trick of on one hand converting all control dependencies in data dependencies while still avoiding unnecessary evaluation, cannot be achieved in a single loop. If one avoids all branches (the "compute-all" approach in Algorithm 3), all conditions always get evaluated eagerly, wasting resources if a prior condition already failed. One can try mixed approaches, branching on the first predicates and using data dependency on the remaining ones. They perform better in some selectivity ranges, but maintain the basic problems – their worst behavior is when the selectivity is around 50%.

Figure 3.4 shows that JIT compilation of conjunctive Select is inferior to the pure vectorized approach. The lazy compiled program (line "C") does slightly outperform vectorized branching (line "A"), but for the medium selectivities the non-branching vectorized implementation (line "B") is by far the best choice. Other variants of compiled implementation (lines "D", "E" and "F"), which balance between being lazy but branching or avoiding branches but evaluating the conditions eagerly, are not much better.

Comparing the best vectorized implementation with the best compiled program for a given selectivity, loop compilation was better only when the selectivity of the condition was below 10% or above 90%. Also, having each operation as a separate vectorized primitive makes it possible to dynamically adapt during runtime based on changes in the selectivity of individual conditions. We can switch between branching and non-branching implementations or reorder the conditions to always evaluate that with the least selectivity first. If we compile the whole selection into a single function evaluating the whole condition in a single loop iteration we lose this flexibility.

Therefore, applying JIT compilation to conjunctive selection conditions has limited usability.

3.4. Case study: Disjunctive select

The next test case is very similar to the previous one from Section 3.3, but with a disjunction of conditions instead of a conjunction.

```
SELECT ...
WHERE col1 >= v1 OR col2 >= v2 OR col3 >= v3
```

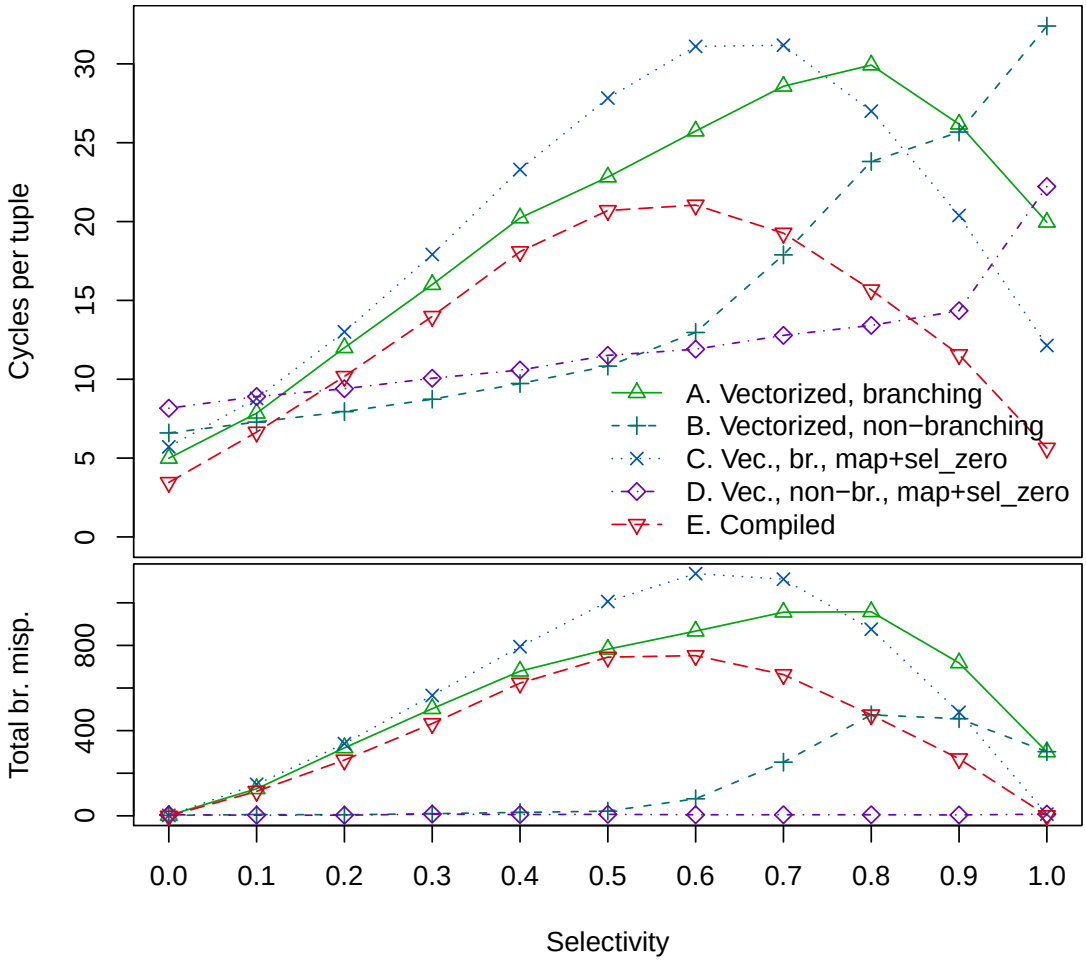


Figure 3.5: Disjunction of selection conditions: total cycles and branch mispredictions vs. selectivity

3.4.1. Vectorized execution

A disjunction of conditions is more complicated for the vectorized computation than a conjunction. If a disjunction is to be performed lazily, the second condition should be evaluated only on those tuples for which the first was false, and the third only on those where both the first two conditions were false. This would require two selection vectors: one for selected and one for not-selected tuples from the first three conditions. Later, a union of the selected tuples would have to be computed. A simpler solution would be to invert the condition using de Morgan's law:

$$a \vee b \vee c = \neg(\neg a \wedge \neg b \wedge \neg c)$$

This way the selection is actually evaluated as a negated conjunction, equivalent to

```
SELECT ...
WHERE NOT (col1 < v1 AND col2 < v2 AND col3 < v3)
```

The selectivity Figure 3.5 is the selectivity of *each* of the *inverted* conditions (i.e. the ones that are actually calculated as a conjunction). So when the selectivity in the figure is x, the total selectivity of the disjunction is $1 - x^3$.

Algorithm 4 Implementations of disjunctive select. The vectorized implementation inverts the disjunction into a conjunction in order to evaluate it lazily. Therefore in the end it has to invert it's result. The first approach, used in VectorWise inverts the final selection vector using a special `diff_selected` primitive, which is prone to branch mispredictions. An alternative method has been implemented, in which the last predicate is evaluated as a map condition, creating a bitmap of zeroes and ones. Then, a selection vector with positions of zeroes in the bitmap can be created while avoiding branches. Compiled implementation just straightforwardly checks the disjunction.

```
const uidx LENGTH = 1024;
uint col1[LENGTH], col2[LENGTH], col3[LENGTH], v1, v2, v3; // Input
uidx tmpsel[LENGTH]; uchr tmpuchr[LENGTH]; // Intermediates
uidx res[LENGTH]; // Resulting selection vector.

/* Special primitive for inverting a selection vector */
uidx diff_selected(uidx n, uidx*__r src, uidx m, uidx*__r dst /*, uidx*__r sel */) {
        uidx i,j;
        for(j=i=0; i<m; i++,j++)
            while(j < src[i]) *dst++ = j++;
        while(j < n) *dst++ = j++;
        return n-m;
}

/* Vectorized implementation, inverting selection vector at the end. (lines ''A'' and ''B'') */
uidx n1 = sel_lt_sint_col_sint_val(LENGTH, tmpsel, col1, val1, NULL);
uidx n2 = sel_lt_sint_col_sint_val(n1, tmpsel, col2, val2, tmpsel);
uidx n3 = sel_lt_sint_col_sint_val(n2, tmpsel, col3, val3, sel2);
uidx retn = diff_selected(n, res, n3, tmpsel, NULL);

/* Vectorized implementation, performing the last selection as a map. (lines ''C'' and ''D'') */
n1 = sel_lt_sint_col_sint_val(LENGTH, tmpsel, col1, val1, NULL);
n2 = sel_lt_sint_col_sint_val(n1, tmpsel, col2, val2, tmpsel);
memset(tmp1, 0, VECLENGTH*sizeof(uchr)); /* Have to zero */
map_lt_sint_col_sint_val(n2, tmpuchr, col3, val3, sel2);
uidx retn = sel_zero_uchr_col(n, res, tmp1, sel);

/* Compiled implementation. (line ''E'') */
uidx sel_compiled(uidx n, uidx* res, uint* col1, uint* col2, uint* col3,
                                     uint* val1, uint* val2, uint* val3)
    for(int i=0, int j=0; i<n; i++)
        // all predicates branching ("lazy")
        if (col1[i] >= v1 || col2[i] >= v2 || col3[i] >= v3)
            res[j++] = i;
    return j;
}
```

The result of a conjunction from previous example is a selection vector. To negate it, a vector of indexes that were not in the resulting selection vector has to be constructed. The vector has to be inverted.

The way VectorWise does it is shown in the top of Algorithm 4. A special primitive `diff_selected` is called at the end, which iterates the selection vector from the conjunction, and for every gap between the indexes in it, output a list of indexes in that gap. When the selection vector is nearly full, so the gaps are small, this results in many very short executions of inner loops. This causes bad performance due to branch mispredictions. It can be seen in Figure 3.5, that the vectorized implementation using data dependency (non-branch) suffers a steep rise in execution cycles when the selectivity of the conditions gets higher (line "B"). This slowdown and rising number of branch mispredictions is the fault of this special `diff_selected` primitive.

Algorithm 5 All-compiled Selection and Projection.

```
uidx selmap_compiled(uidx n, uint *res, uidx *ressel, uint *price, uchr* tax, uchr* discount,
                     uint *col1, uint *v1) {
    for(int i=0, int j=0; i<n; i++)
        if (col1[i] < v1) {
            res[i] = price[i]*((100-discount[i])*(100+tax[i]));
            ressel[j++] = i;
        }
    return j;
}
```

An alternative way would be to calculate the very last operation using a map operation instead of a selection, on a previously zero-initialized vector, as is also presented in Algorithm 4. The map operation works on a selection vector resulting from the conjunction of previous conditions, and it puts "ones" in the map result vector only where also the last condition is true. Then zeroes can be selected from the map result, which can be done without branching with a standard `sel_nonzero` primitive. This implementation does not suffer from branch mispredictions at high selectivities, and its data dependency variant is the best on a very wide range of selectivities and only slightly outperformed on other (lines "C" and "D") .

As previously, VectorWise chooses either the branch or non-branch strategy depending on the observed selectivity. Therefore it can achieve the minimum from the branching and non-branching lines in Figure 3.5.

3.4.2. Loop-compiled execution

The compiled implementation of a disjunctive Select is straightforward, evaluating the conditions in a branching `if`. The performance of the compiled implementation does not differ much from that from experiment in Section 3.3. Since a disjunction is harder for the vectorized implementations, JIT compilation manages to compete better with them. Nevertheless, the range of selectivities at which it is better than a vectorized implementation is only a little wider than that for a conjunctive selection. The conclusion from Section 3.3 and this one is that it is in most cases non profitable to introduce JIT compilation of selections in VectorWise.

3.5. Synthesis: Selection and Projection

We discussed Projections in Section 3.2 and Selections in Sections 3.3 and 3.4. We can now demonstrate a synthesis of both cases, analyzing the following query:

```
SELECT l_extprice*(1-l_discount)*(1+l_tax)
FROM lineitem
WHERE col1 < val1;
```

The all-compiled generated code shown in Algorithm 5 has all of the single-loop compilation problems described in previous sections: evaluation of the condition causes branch mispredictions and conditional computation of the selection prevents it from being SIMDized.

In Figure 3.6 we show how applying various described optimizations makes the vectorized implementation superior to to the all compiled implementation is shown in line "A". Not surprisingly, a plain implementation using branching selection and computing the projection

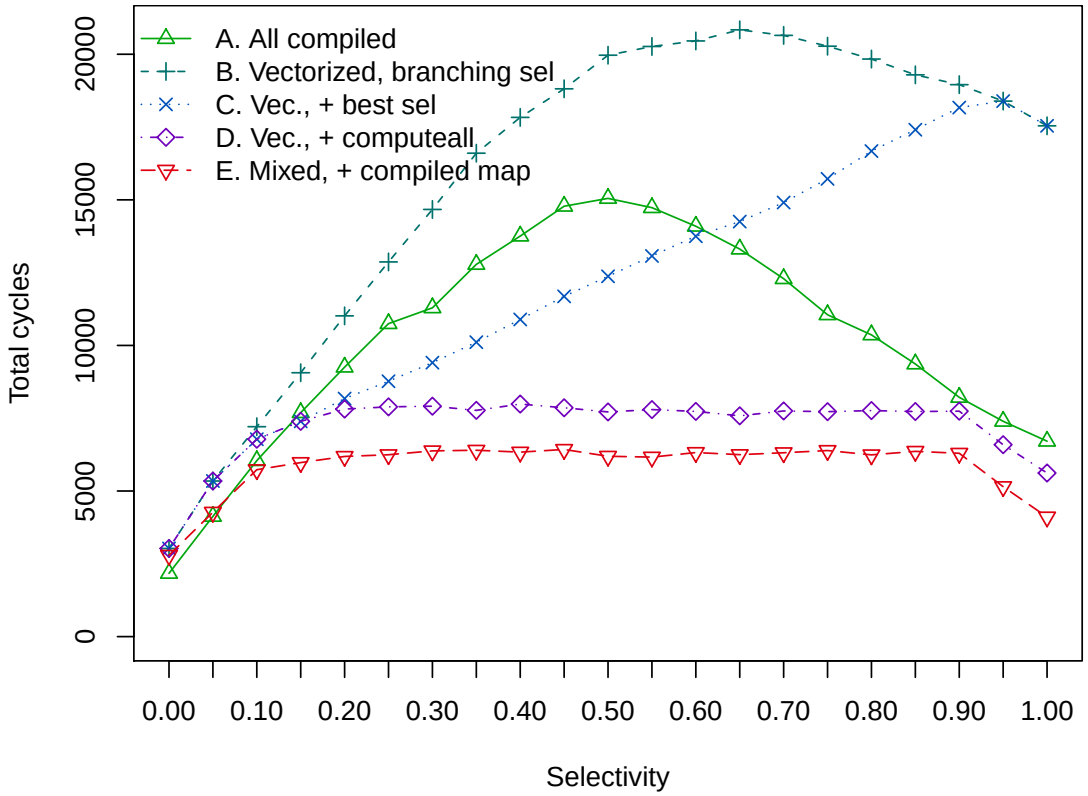


Figure 3.6: Selection and projection. The all-compiled implementation is compared to vectorized implementations with different optimizations described in previous Sections.

with a selection vector (line "B") is the slowest. However choosing between branching and non-branching selection (line "C"), as described in Section 3.3.1, already makes the vectorized implementation competitive with the all-compiled. Adding the compute-all optimization from Section 3.2.6 (line "D") makes vectorized implementation better than all-compiled for almost all selectivities. The final optimization (line "E") is to compile Project together in single-loop. With an auto-SIMDizing compiler, the loop will get optimized using SSE and the compute-all optimization can be applied to it. Section 3.2.6 has shown it to be the fastest Projection implementation.

This example demonstrated a case where single-loop compilation is beneficial, but compiling too much can hurt performance rather than improve it, so partial compilation is the best answer.

3.6. Case Study: Hash Join

Our last micro-benchmark is related Hash Joins. We use the following SQL query:

```
SELECT build.col1, build.col2, build.col3
WHERE probe.key1 = build.key1 AND probe.key2 = build.key2
FROM probe, build
```

We focus on an equi-join condition involving keys consisting of two integer columns, because such composite keys are more challenging for vectorized execution. This discussion assumes

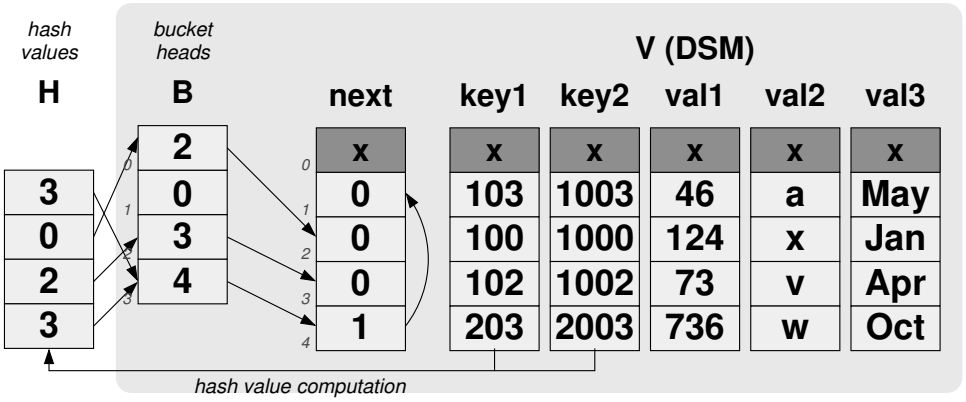


Figure 3.7: Bucket-chain hash table as used in VectorWise. The value space V presented in the figure is in DSM format, with separate array for each attribute. It can also be implemented in NSM, with data stored tuple-wise. From [Zuk09].

simple bucket-chaining such as used in VectorWise, presented in Figure 3.7. This means that keys are hashed into buckets in a bucket array B, whose size N is a power of two. The bucket contains the offset of a tuple in a value space V. This space can either be organized using DSM or NSM layout; VectorWise supports both [ZNB08]. It contains the values of the build relation, as well as a next-offset, which implements the linked list. The value space is therefore either a collection of arrays, one for each column of the build relation and the next-offset (DSM), or one big array of tuples (NSM). A bucket may have a chain of length greater than one either due to hash collisions, or because there are multiple tuples in the build relation with the same keys.

The bottleneck in hash table operations are random accesses to huge arrays in memory. Those accesses not only cause cache misses, but also, which is often overlooked, can cause a mayhem of TLB misses, which make the performance an order of magnitude slower than that of the experiments from previous sections.

We measured the performance of both building and probing phase of a hash join 0-1 (with at most one tuple in the build relation matching a tuple in the probe relation). In three experiments, we vary hash table value space V size (the build relation size), selectivity (fraction of probe keys that match something), and bucket array B size (which influences the number of collisions and therefore length of bucket chains). The default values are 16 M tuples in the value space, number of buckets equal to the value space size and match rate 100% – in the experiments we vary one of the values, when the other two remained fixed. The size of the value space was changed from 4 K to 64 M in the first experiment, match rate from 0% to 100% in the second and buckets number from 2 times the value space size to $\frac{1}{32}$ times the value space size in the third.

Figure 3.8 shows the building phase of hash join. Figure 3.9 shows the probing phase of hash join. The left graph shows that when the hash table size grows, performance deteriorates; it is well understood that cache and TLB misses are to blame, and DSM achieves less locality than NSM. The middle graph shows that with increasing match rate the cost goes up, which is mainly caused by more tuples needed to be fetched. The compiled NSM fetch performs best, as will be explained in Section 3.6.5. The right graph shows what happens with increasing chain length. The fully compiled variant suffers most, as it gets no parallel memory access, as will be explained in Section 3.6.4.

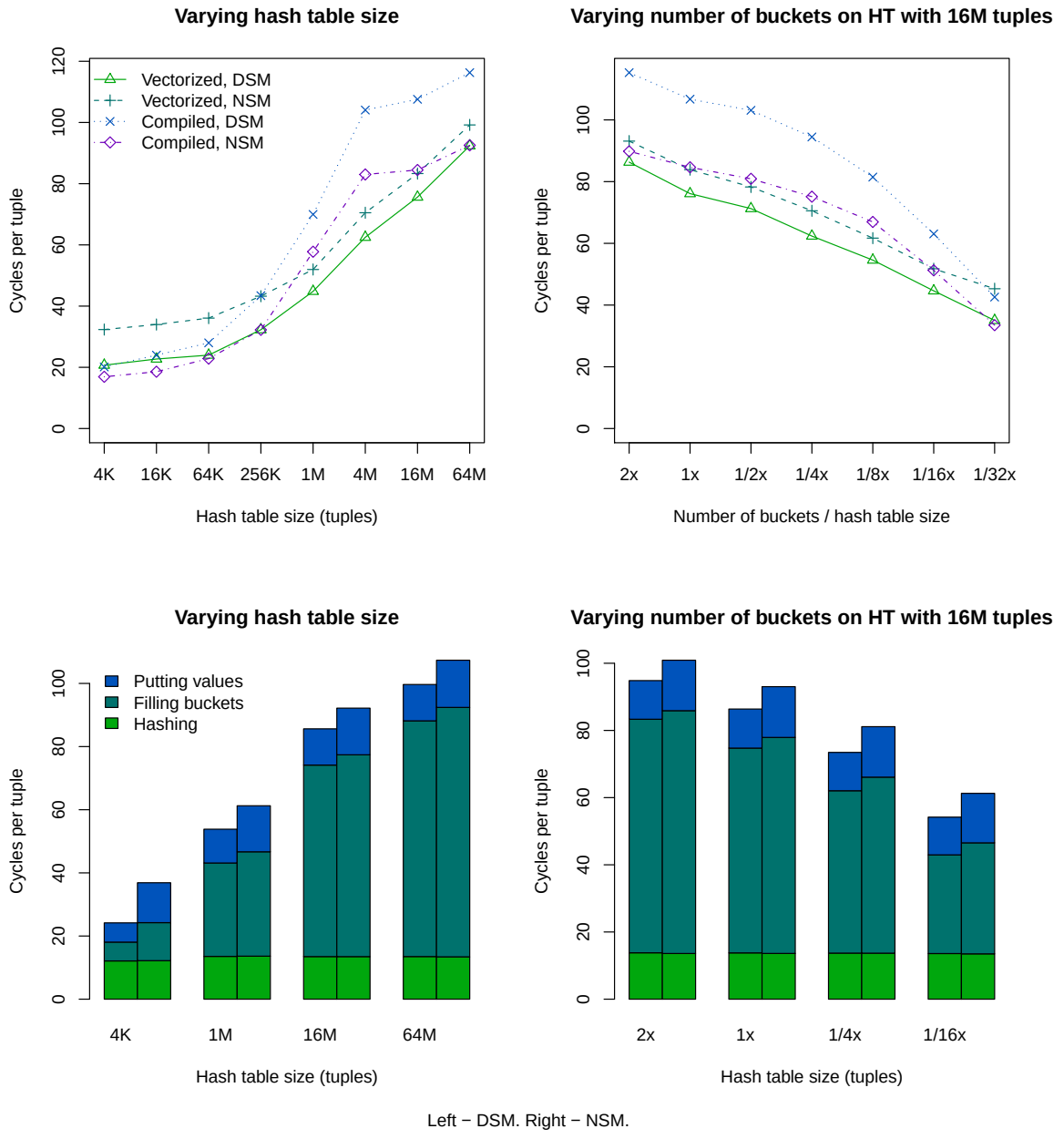


Figure 3.8: Hash table building. Left: different sizes of the hash table value space V. Right: different sizes of the bucket array B w.r.t. value space V (different chain lengths). Fixed parameters are: V size 16M tuples (448 MB), selectivity 100%, B array size equal to V size.

3.6.1. Vectorized implementation

Algorithm 7 contains high level pseudocode of hash table building and probing. Additionally, Algorithm 6 shows (simplified) C code for the primitives that are used in Algorithm 7.

Building starts by vectorized computation of the hash number from the key in a column-by-column fashion using map-primitives. A `map_hash` primitive first hashes each key of type `T` onto a `lng` long integer. If the key is composite, we iteratively refine the hash value using a `map_rehash` primitive, passing in the hash values previously computed and the next key

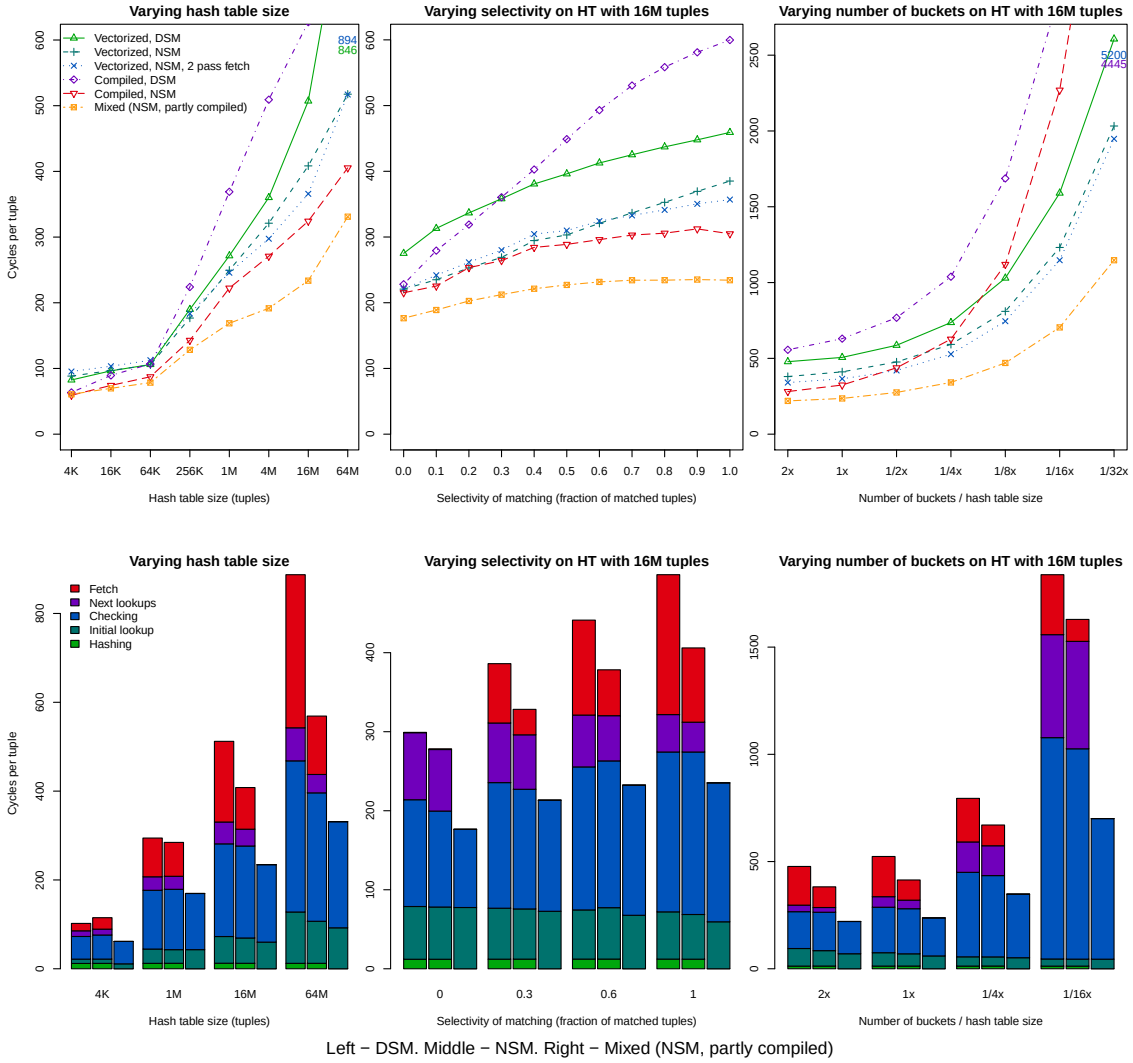


Figure 3.9: Hash table probing. Left: different sizes of the hash table value space V. Middle: different match rates of the probe. Right: different sizes of the bucket array B w.r.t. value space V (different chain lengths). For the partially compiled implementation, hashing and lookup initial are together as lookup initial, and checking with lookup next and fetching are together as checking. Fixed parameters are: V size 16M tuples (448 MB), selectivity 100%, B array size equal to V size.

column. A bitwise-and map-primitive is used to compute a bucket number from the hash values: $H\&(N\text{-}1)$.

Sequential free slots at the back of the value space V are allocated for the inserted tuples. A tuple is made the new head of the chain in the computed bucket of B, and the previous head is saved in the "next" field of V. This is done by a special `ht_insert` primitive.

Probing starts by vectorized computation of a hash number and bucket number exactly like in building. To read the positions of heads of chains for the calculated buckets we use a special primitive `ht_lookup_initial`. It behaves like a selection primitive, creating a selection vector $\vec{Match}$ of positions in the bucket positions vector $\vec{H}$ for which a match was found in bucket array B. Also, it fills the $\vec{Pos}$ vector with positions of the candidate

Algorithm 6 Primitives used by the vectorized implementation of hash table operations. Exact format of primitive parameter may vary, depending on whether value space is in DSM or NSM storage.

```
// Primitive for hashing, first column
map_hash_T_col(uidx n, ulng* res, T* col1) {
    for(uidx i=0; i<n; i++)
        res[i] = HASH(col1[i]);
}

// Primitive for hashing, subsequent columns
map_rehash_ulng_col_T_col(uidx n, ulng* res, ulng* col1, T* col2) {
    for(uidx i=0; i<n; i++)
        res[i] = col1[i] ^ HASH(col2[i]);
}

// Primitive for fetching values.
map_fetch_uidx_col_T_col(uidx n, T* res, uidx* col1, T* base, uidx* sel) {
   if(sel) {
       for (idx i=0;i<n; i++)
            res[sel[i]] = base[col1[sel[i]]];
   } else {/* sel == NULL,., omitted */}
}

// Primitive for fetching and checking a key, first column.
map_check_T_col_idx_col_col_T_col(idx n, chr* res, T* keys, T* base, idx* pos, idx* sel) {
   if(sel) {
       for(idx i=0; i<n; i++)
            res[sel[i]] = (keys[sel[i]]!=base[pos[sel[i]]]);
   } else {/* sel == NULL, omitted */}
}

// Primitive for fetching and checking a key, subsequent columns.
map_recheck_chr_col_T_col_idx_col_T_col(idx n, chr* res, chr* col1,
                                        T* keys, T* base, idx* pos, idx* sel) {
    if(sel) {
        for(idx i=0; i<n; i++)
            res[sel[i]] = col1[sel[i]] || (keys[sel[i]]!=base[pos[sel[i]]]);
    } else {/* sel == NULL, omitted */}
}

// Primitive for inserting into hash table.
ht_insert(uidx n, uidx* H, uidx* B, ValueSpace* V) {
    for(uidx i=0; i<n; i++) {
        V->next[V->first_empty] = B[H[i]]; /* moving old chain head to ,,next'' */
        B[H[i]] = V->first_empty; /* saving new chain head */
        V->first_empty++;
    }
}

// Primitive for looking up the head of bucket chain.
ht_lookup_initial(uidx n, uidx *pos, uidx* match, uidx* H, uidx* B) {
    for(uidx i=0,k=0; i<n; i++) {
        pos[i] = B[H[i]]; /* saving found chain head position in V */
        match[k] = i; k += (pos[i]!=0); /* saving to a sel. vector if non-zero */
    }
}

// Primitive for advancing to next element in a bucket chain.
ht_lookup_next(uidx n, uidx *pos, uidx* match, ValueSpace* V) {
    for(uidx i=0,k=0; i<n; i++) {
        pos[match[i]] = V->next[pos[match[i]]]; /* advancing to next in chain */
        match[k] = match[i]; k += (pos[match[i]]!=0); /* to a sel. vec. if non-zero */
    }
}
```

Algorithm 7 Pseudocode for the operator level logic of hash table operations. Primitives from Algorithm 6 are used.

```
procedure HTBUILD(V, B[1..N], K⃗_{1..k}, V⃗_{1..v})
    H⃗ ← map_hash(K⃗_1)                                          ▷ Hashing
    for each remaining key vector K_i do
        H⃗ ← map_rehash(H⃗, K⃗_i)
    H⃗ ← map_bitwiseand(H, N − 1)
    first_put_pos ← V.first_empty                               ▷ Save position where to start putting
    ht_insert(H⃗, B, V)                                         ▷ Filling buckets with new chain heads
    for each x⃗ in K⃗_i and V⃗_i do                               ▷ Putting
        Put x⃗ into V starting at first_put_pos

procedure HTPROBE(V, B[0..N − 1], K⃗_{1..k}(in), R⃗_{1..v}(out), Hits⃗(out))
    H⃗ ← map_hash(K⃗_1)                                          ▷ Hashing
    for each remaining key vectors K_i do
        H⃗ ← map_rehash(H⃗, K⃗_i)
    H⃗ ← map_bitwiseand(H, N − 1)
    Pos⃗, Match⃗ ← ht_lookup_initial(H, B)                       ▷ Initial Lookup
    while Match⃗ not empty do
        Check⃗ ← map_check(K⃗_1, V_{key_1}, Pos⃗, Match⃗)          ▷ Checking
        for each remaining key vector K⃗_i do
            Check⃗ ← map_recheck(Check⃗, K⃗_i, V_{key_i}, Pos⃗, Match⃗)
        Match⃗ ← sel_nonzero(Check⃗, Match⃗)
        Pos⃗, Match⃗ ← ht_lookup_next(Pos⃗, Match⃗, V_next)       ▷ Chain Following
    Hits⃗ ← sel_nonzero(Pos⃗)
    for each result vector R⃗_i do                               ▷ Result Fetching
        R⃗_i ← map_fetch(Pos⃗, V_{value_i}, Hits⃗)
```

matching tuples in the hash table. If the value (offset) in the bucket is 0, there is no key in the hash table – these tuples store 0 in $\vec{Pos}$ and are not part of $\vec{Match}$.

Having identified the positions of possible matching tuples, the next task is to "check" if the key values actually match. This is implemented using a specialized map primitive that combines fetching a value from the given offset of value space V with testing for non-equality: `map_check`. Similar to hashing, composite keys are supported using a `map_recheck` primitive which gets the boolean output of the previous check as an extra parameter. The resulting booleans mark positions for which the check failed. These positions can then be selected using a select primitive, overwriting the selection vector $\vec{Match}$ with positions for which probing should advance to the "next" position in the chain. Moving on in the linked lists is done by a special primitive `ht_lookup_next`, which for each probe tuple in $\vec{Match}$ fills $\vec{Pos}$ with the next position in the bucket-chain of V. It also guards ends of chain by saving in $\vec{Match}$ only the subset of positions for which the resulting position in $\vec{Pos}$ was non-zero.

The loop finishes when the $\vec{Match}$ selection vector becomes empty, either because of reaching end of chain (element in $\vec{Pos}$ equals 0, a miss) or because checking succeeded (element in $\vec{Pos}$ pointing to a position in V, a match).

Matches can be found by selecting the elements of $\vec{Pos}$ which ultimately produced a hit with a `sel_nonzero` primitive. $\vec{Pos}$ with selection vector $\vec{Hits}$ becomes a pivot vector for fetching. This pivot is subsequently used to fetch (non-key) result values from the build value area into the hash join result vectors; using one fetch primitive for each result column. A basic fetching primitive is shown in Algorithm 6, but an in-depth description of fetching will be provided in Section 3.6.5.

The primitives `map_recheck`, `ht_lookup_initial` and `ht_lookup_next` use branching instructions that could be avoided with a data dependency. However, the performance of those is dominated by TLB misses they cause, not by branch mispredictions. It was tested, that

Algorithm 8 Fully loop-compiled hash operations, working just like vectorized implementation would for vectors of size 1.

```
HTprobe_build(uidx n, ValueSpace* V, uidx* B, T* key1, T* key2, T* val1, T* val2, T* val3) {
    for(uidx i = 0; i < n; i++) {
        uidx pos = V->first_empty, hash = (HASH(key1[i])^HASH(key2[i]))&(N-1);
        V->next[pos] = B[hash]; B[hash] = pos;
        V->key1[pos] = key1[i]; V->key2[pos] = key2[i];
        V->col1[pos] = val1[i]; V->col2[pos] = val2[i]; V->col3[pos] = val3[i];
        V->first_empty++;
    }
}

HTprobe_comp(uidx n, ValueSpace* V, uidx* B, T* key1, T* key2, T* res1, T* res2, T* res3) {
    for(uidx i=0; i<n; i++) {
        uidx pos, hash = HASH(key1[i])^HASH(key2[i]);
        if (pos = B[hash&(N-1)]) do {
            if(key1[i]==V[pos].key1 && key2[i]==V[pos].key2) {
                res1[i] = V->col1[pos]; res2[i] = V->col2[pos];
                res3[i] = V->col3[pos]; break; // found match
            }
        } while(pos = V->next[pos]); // next
    }
}
```

changing those branches into data dependency made no difference to the results.

Figures 3.8 and 3.9 feature bar graphs splitting the execution time of the vectorized implementation into phases annotated on the right margin of Algorithm 7. Such fine-grained profiling is possible because those steps are performed for whole vectors of data. In the compiled implementations, the overhead of starting and stopping the counters after operations on single tuples would be to big to get any meaningful measurements.

3.6.2. Loop-compiled implementation

It is possible to create a loop-compiled version of the whole join, like proposed in e.g. HIQUE [KVC10]. Both the build and the probe phase work much like vectorized implementation would for vectors of size 1. Algorithm 8 shows the code of such an implementation.

It can be seen from Figure 3.9 that overall the compiled implementation using NSM hash table is faster than the vectorized ones. However, it will be discussed in the following sections, that it has its shortcomings.

3.6.3. Build phase

Figure 3.8 shows that different implementation of hash table building have similar performance. All implementations of build phase have sequential access patterns to the value space V. The only difference between vectorized or compiled implementation and DSM or NSM layout of the value space V is either fully sequential access to one array, or a series of sequential accesses with a stride, or many interleaved sequential cursors. All these methods nevertheless operate with good locality and linearity on a small portion of value space V, which should become cache resident and do not cause cache or TLB misses.

The phase-split part of Figure 3.8 shows that the dominant part of building the hash table is filling the buckets arrays B. This is because it involves accesses to random positions in the arrays, therefore causing TLB misses.

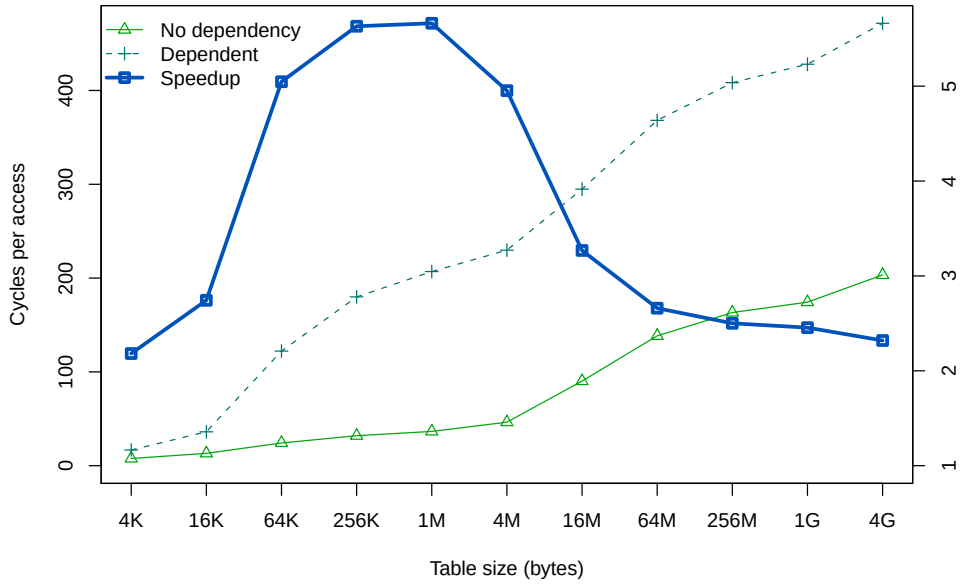


Figure 3.10: Dependent and independent memory accesses

3.6.4. Probe lookup: bucket chain traversal

The right graph of Figure 3.9 shows an experiment in which we decreased the number of buckets (size of array B) which results in more hash collisions, and therefore longer chains that have to be followed during lookup. Results of this show that performance of the compiled implementations degrades dramatically when the number of buckets decreases (and so the collision lists become longer). This leads to a conclusion that the bucket chain traversal phase in the compiled implementation must be slower than in the vectorized one. To explain it, we investigate how memory accesses work on modern processors.

Because memory latency is not improving much (∼50ns), and cache line granularities must remain limited (64bytes), memory bandwidth on modern hardware is no longer simply the division between these two. A single Nehalem core can achieve a factor 10 more than this 1.28GB/s thanks to automatic CPU prefetching on sequential access. Performance thus crucially depends on having multiple outstanding memory requests at all times. For random access, this is hard to achieve, but the deeply pipelined out-of-order nature of modern CPUs does help. That is, if a load stalls, the CPU might be able to speculate ahead into upstream instructions and reach more loads. Speculation-friendly code is therefore more effective than manual prefetching, which tends to give only minor improvement, and is hard to tune/-maintain for multiple platforms. Success is not guaranteed, since the instruction speculation window of a CPU is limited, depends on branch prediction, and only independent upstream instructions can be taken into execution.

A small experiment demonstrates the difference. Two code snippets are shown in Algorithm 9. In the first case, the memory accesses are independent. In the second one, previous memory access has to be completed to get the new value of `k` before the next one can be issued. The input is prepared in such a way, that `t[in[k]]] = k+1`, so that always $k = i$, and so both procedures perform the same sequence of reads from a big array in memory – `t[in[0]], t[in[1]], t[in[2]], ...`.

Despite that, Figure 3.10 shows that the procedure with independent accesses works up to 5 times faster, because the processor can dispatch memory accesses and handle the misses they

Algorithm 9 Dependent and independent memory accesses. The accessed memory positions are exactly the same, but in the first example they are independent, while in the second one previous access has to be completed before proceeding to the next one.

```
// in: random positions in t.
void nodep(int n, int *r, int *in, int *t) {
    for(int i = 0; i < n; i++)
        r[i] = t[in[i]];
}

void dep(int n, int *r, int *in, int *t) {
    // (1.) t[in[k]] == k+1
    for(int i = 0, int k=0; i < n; i++)
        // (2.) k == i always, because of (1.)
        k = r[i] = t[in[k]];
}
```

trigger in parallel, while the dependent procedure can perform and complete only one memory access at a time. The crux here is that the vectorized primitive trivially achieves whatever maximum outstanding loads the CPU supports, as it is a tight loop of independent loads. The fully compiled hash probe from Algorithm 8 however, can run aground while following the bucket chain. Its performance is only good if the CPU can speculate ahead across multiple probe tuples (execute concurrently instructions from multiple for-loop iterations for different `i`). That depends on the branch predictor predicting the `while(pos)` to be false, which will happen in join key distributions where there are no collisions and the chains stay short. If however the build relation has long chains of tuples in one bucket, the CPU will stall with a single outstanding memory request (`V[pos]`, because the branch predictor will make it stay in the while-loop and it will be unable to proceed, as the value of `pos = V.next[pos]` is unknown because it must wait for the resolution of the current cache/TLB miss. This causes fully compiled hashing to suffer a dramatic slowdown when the bucket chains are long, as shown by lines "D" and "E" in Figure 3.9.

3.6.5. Probe fetching

The phase of hash table probing in which we fetch the values of matched tuples into output vectors deserves a separate discussion. The basic vectorized fetching implementation was shown in Algorithm 6. A vectorized fetching operation takes a pivot vector $\vec{Pos}$ of indexes of the matched tuples in the hash table value space V. The vectorized operation fetches values one column at a time. Since $\vec{Pos}$ contains random positions in a huge data array, each access is expected to cause a TLB miss.

If V is stored in DSM, then the different columns are stored in separate arrays and nothing can be done – fetching the pivot positions from different columns again triggers TLB misses for that new memory regions. However, if the V is in NSM, the values from different columns are adjacent to each other. Yet, after a primitive finishes fetching a whole vector of values for the first column and proceeds to the next one, tuples in V at positions from the beginning of the pivot vector would already be evicted from the cache and TLB. Figure 3.12 demonstrates this. The TLB cache size on Nehalem processors has been significantly increased compared to older architectures, with 512 entries on the second level of TLB. As soon as the pivot vector gets longer than this, performance drops dramatically and an increased number of TLB misses is observed.

Obviously, a compiled solution, in which a primitive is generated for a given combination of attributes, fetching them all at once from an NSM tuple directly into a number of output

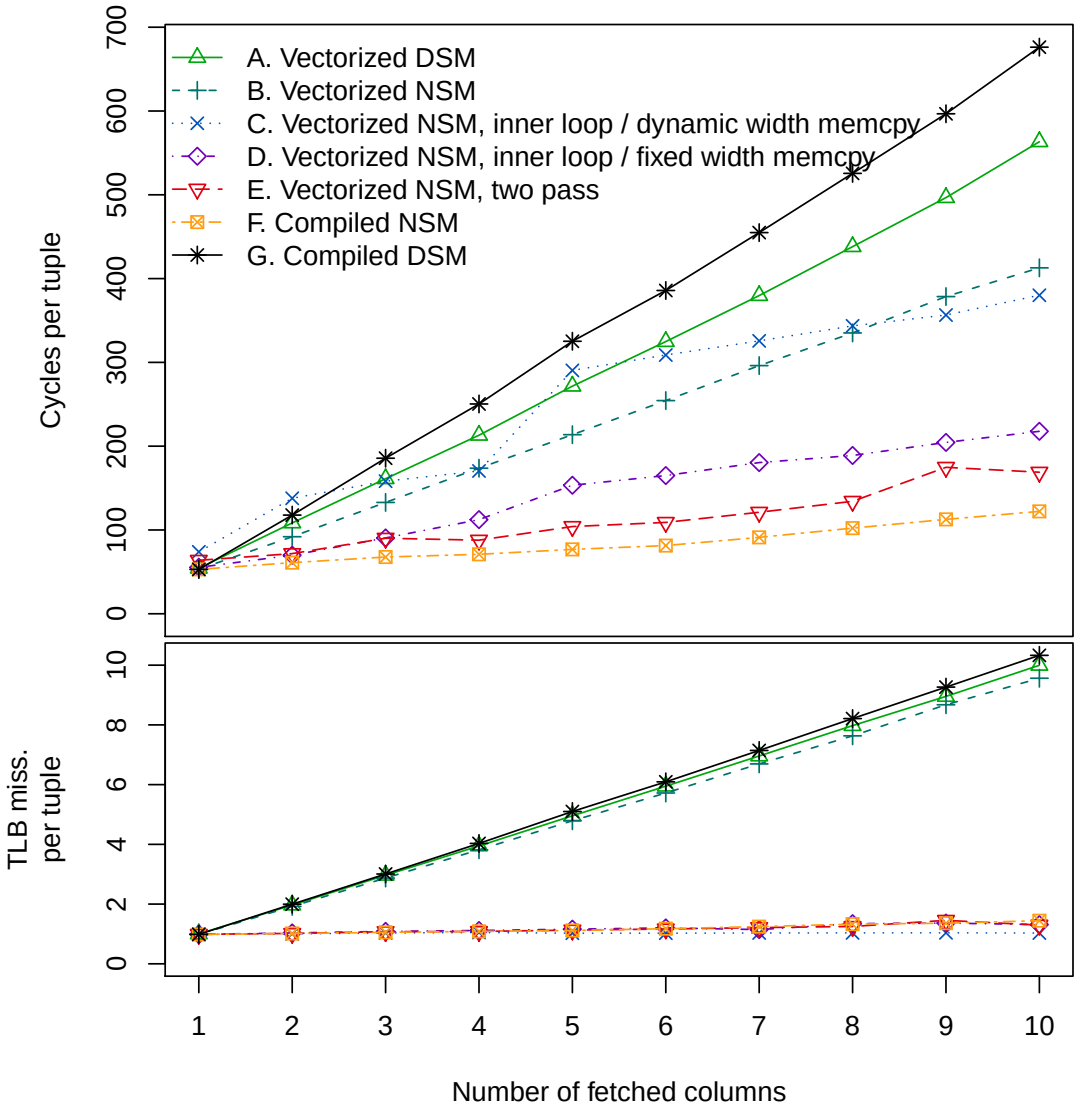


Figure 3.11: Fetching from 1 to 10 columns of data from a hash table value space using different methods: cycles per tuple and total TLB misses.

vectors will be superior. The reason for using JIT is the number of possible combinations of types of columns and their number is too high, and specialized primitives could not be statically pre-generated for all of them.

There are however other ways in which the memory access pattern of a vectorized primitive can be improved without using compilation. Algorithm 10 shows different vectorized methods of fetching data from a hash table. The first two are the already discussed simple fetching from DSM (line "A") and NSM (line "B") value space. The third method tries to achieve the flexibility of a compiled solution in an interpreted way. The "inner loop" fetch primitive takes the description of columns and their offsets in an NSM space as parameters, and uses a nested inner loop to copy out all columns of a tuple at once into given output vectors. All types of data can be and copied out using `memcpy` library call, knowing only the datatype width. When widths of datatypes are given as parameters to `memcpy`, such a call constitutes

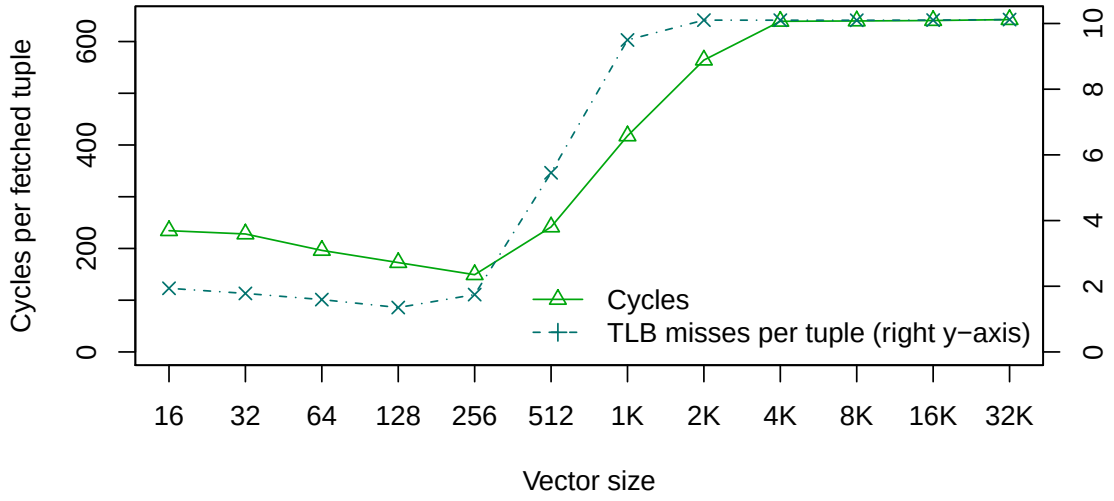


Figure 3.12: Fetching from an NSM value space column by column with different vector lengths.

an additional cost. However, if the width of `memcpy` is known at compile time, the compiler is able to inline and optimize it, often into a simple memory-register-memory move.

If it is safe to write to a few bytes of memory past the end of a vector (e.g. by overallocating) a width greater than the actual width of the data type can safely be used. We are able to statically pre-generate primitives that always use `memcpy` with a compile-time constant value of `WIDTH`, and the `widths` parameter is used only for calculating offsets. The static primitive with the `WIDTH` equal to the width of the widest attribute in the tuple may be used, as the thinner attributes can safely be copied with a few bytes of redundant padding. It can be seen in Figure 3.11 that while the implementation with dynamic width of `memcpy` (line "C") does not give much improvement compared to the basic vectorized fetching, the fixed `memcpy` (line "D") performs significantly better. The cost is sometimes copying more than strictly necessary and the need of a few bytes of additional space overallocated after the end of the vectors.

The "inner loop" version introduced more complicated code, and inner loop and interpretation overhead. The last, "two pass", implementation (line "E") uses a different approach. It is more in the "vectorized spirit": use intermediate results that allow to use code with simple tight loops. The method works in two passes: first copies whole tuples from an NSM value space into an intermediate contiguous NSM space, and then converts the local NSM buffer into DSM output, avoiding TLB and cache misses because of locality. As results in Figure 3.11 show, even though additional memory traffic is needed, the simpler construction of this implementation makes it perform better than "inner loop".

The compiled implementation is still the fastest one, but when the optimized vectorized implementations are taken into account, the difference is not so big. While the compiled NSM implementation (line "F") is around 5 times faster than the basic vectorized one used in VectorWise when fetching 10 columns of data, the speedup over the best vectorized "two pass" approach is only 28%. A vectorized implementation of hash join probing using the "two

Algorithm 10 Simplified code of different vectorized implementations of fetching values from a hash table value space V. The `stride` parameter is the width of the tuple in an NSM value space. By using a `stride` and an offset we can access individual attributes in the tuples.

```
// 1. Simple fetch from a DSM value space.
map_fetchDSMtoDSM_uidx_col_T_val(uidx n, T* res, uidx* pivot, T* base) {
    for(uidx i=0; i<n; i++)
        res[i] = base[pivot[i]];
}

// 2. Simple fetch from a NSM value space.
map_fetchNSMtoDSM_uidx_col_T_val_uidx_val(uidx n, T* res, uidx* pivot, T* base, uidx stride) {
    for(uidx i=0; i<n; i++)
        res[i] = base[stride*pivot[i]];
}

// 3. Fetching all columns at once from NSM in an inner loop, two variants.
map_fetchINNERLOOP(uidx n, uidx stride, uidx *widths, void **res, uidx* pivot, void* base) {
    for(uidx i=0; i<n; i++) {
        void *source = input+WIDTH*pivot[i]*stride;
        for(uidx k = 0; k < ncols; k++) {
            // 3a. dynamic width memcpy:
            memcpy(res[k]+widths[k]*i, source, widths[i]);
            OR
            // 3b. compile-time fixed width memcpy:
            memcpy(res[k]+widths[k]*i, source, WIDTH);

            source += widths[k]; // next offset
        }
    }
}

// 4. Two pass fetching.
map_fetchNSMtoNSM(uidx n, void* res, uidx* pivot, void* base, uidx stride) {
    for(uidx i=0, i<n; i++)
        memcpy(res+i*stride, base+pivot[i]*stride, stride);
}
map_fetchDIRECT_T_val_uidx_val(uidx n, T* res, T* base, uidx stride) {
    for(uidx i=0; i<n; i++)
        res[i] = base[stride*i];
}
fetchTWOPASS(uidx n, uidx ncols, uidx *widths, void **res, uidx* pivot, void* base) {
    /* First pass: copy the pivot tuples into an intermediate contiguous NSM space */
    map_fetchNSMtoNSM(n, tmp, pivot, base, SUM(widths));
    /* Second phase: copy the columns from the temporary space into output vectors */
    for(uidx i = 0; i < ncols; i++) {
        // a map_fetchDIRECT primitive of correct type would be used.
        map_fetchDIRECT_T_val_uidx_val(n, res[i], base, SUM(widths));
        base += widths[i]; // next offset
    }
}
```

pass" approach has also been included in Figure 3.9, and it has also been used for the NSM fetching phase in the phase-split bar graph. To get a full picture, a compiled DSM line ("G") has also been included in Figure 3.11, showing that it has even worse performance than all vectorized ones. This is obvious, because by compiling fetching for a DSM value space we do not achieve anything. With a DSM value space, a vectorized primitive accesses random locations from an array for just one attribute, while a compiled function accesses values from different attributes at once, so the random access pattern is scattered over an even bigger range of memory.

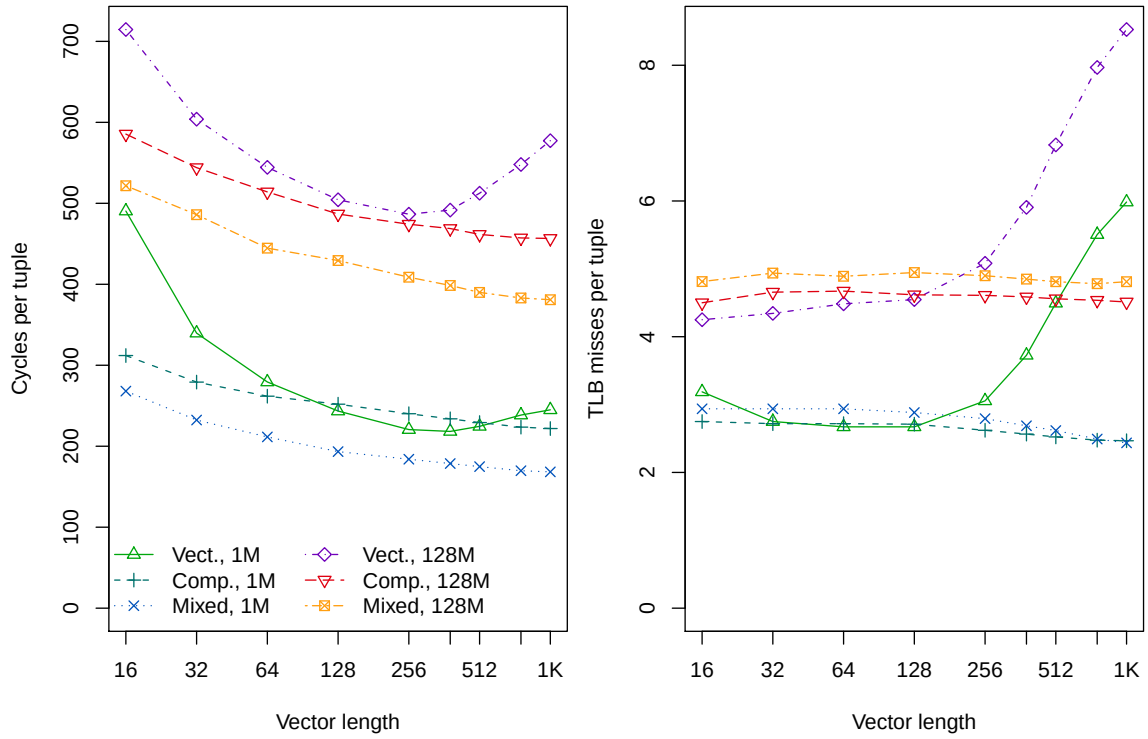


Figure 3.13: Probing of two NSM hash tables: smaller with 1 M tuples (24 MB), bigger with 128 M tuples (3 GB), using different lengths of vectors.

3.6.6. Reduced vector length

Section 3.6.5 has shown that vectorized implementation suffers from TLB cache trashing as soon as vectors lengths exceed the size of TLB cache. We check if reducing the vector length can improve the overall performance of probing, just as it improved the performance of fetch phase.

Figure 3.13 shows the results of an experiment, in which two hash tables, small and large, were probed using varying vector lengths. Parameters were like in the other experiments: 100% match rate and the number of buckets equal to the number of tuples. For that combination of parameters the loop-compiled implementation had the biggest advantage over the vectorized one. The length of the vectors does not affect the compiled implementation much, as it is a monolithic function, only once looping over the vector while processing it.

On the smaller hash table, the compiled implementation causes on average a little over 2 TLB misses per tuple. This corresponds to one miss while accessing the bucket array B, another one accessing the tuple, and a fraction corresponding to the odd case of a hash collision and having to advance to the next tuple in the linked list. For small vector lengths, the vectorized implementation performs similarly. For vectors of length 256, when there is still no TLB cache trashing, it reaches its maximal performance which slightly beats the loop-compiled implementation. However, as the vectors get longer, the number of TLB misses gets higher, reaching an average of over 6 misses per tuple. These 4 additional tuples correspond to a TLB miss for the vectorized operation on every column of hash table: checking of the second key column, and fetching of three value columns. Each of these operations accesses the same elements anew, but with long vectors they are already evicted from TLB. The

Algorithm 11 Partially compiled hash probing. ValueSpace V is in NSM..

```
comp_lookup_initial(uidx n, uidx* pos, uidx* match, T* key1, T* key2, uidx* B, uidx N) {
    uidx k = 0, i;
    for(uidx i = 0; i < n; i++) {
        uidx bucket = (HASH(key1[i])^HASH(key2[i]))&(N-1); /* find bucket */
        pos[i] = B[H[i]]; match[k] = i; k += (pos[i]!=0); /* as in ht_lookup_initial */
    }
    return k;
}

comp_check_fetch_next(uidx n, ValueSpace* V, uidx* pos, uidx* match,
                      T* key1, T* key2, T* res1, T* res2, T* res3) {
    uidx k = 0, i;
    for(i = 0; i < n; i++) {
        if(key1[i] != V[pos[match[i]]].key1 | key2[i] != V[pos[match[i]]].key2) {
            /* as in ht_lookup_next */
            pos[match[i]] = V[pos[i]].next;
            match[k] = match[i];
            k += (pos[match[i]]!=0); /* k <= i, does not overwrite ahead */
        } else {
            /* as in fetch */
            res1[match[i]] = V[pos[i]].val1; res2[match[i]] = V[pos[i]].val2;
            res3[match[i]] = V[pos[i]].val3;
        }
    }
    return k;
}
```

procedure HTPROBE($V, B[0..N-1], \vec{K_{1..k}}$**(in)**$, \vec{R_{1..v}}$**(out)**$, \vec{Hits}$**(out)**)

  $\vec{Pos}, \vec{Match} \leftarrow comp\_hash\_rehash\_bitand(\vec{K_{1..k}}, B)$ ▷ Hashing and initial Lookup

  while $\vec{Match}$ not empty **do** ▷ Checking and fetching or chain following combined.

    $\vec{Pos}, \vec{Match}, \vec{R_{1..v}} \leftarrow comp\_check\_fetch\_next(V, \vec{Pos}, \vec{Match})$

  $\vec{Hits} \leftarrow sel\_nonzero(\vec{Pos})$ ▷ Fetching already done, just save the matched positions.

performance in Figure 3.13 for vectors of length 1024 corresponds to that from Figure 3.9, since these are the same parameters.

Probing the bigger hash table is slower and all algorithms make more TLB misses. Since the hash table is much bigger than 2 MB, a TLB miss occurs not only on the pages itself, but also on the directory pages holding the translations, as was explained in Section 2.3.6. When accessing a position in buckets array B or value space V two TLB misses occur: one for the virtual memory page, and one for the directory page. That is why there is on average a little over 4 TLB misses per tuple instead of 2.

This scenario shows that a pure vectorized approach can be competitive with single-loop compiled when the vector lengths are reduced to prevent TLB trashing. Both of them are however always surpassed by a hybrid partially-compiled implementation, presented in Section 3.6.7.

3.6.7. Partial Compilation

We now propose a hybrid partially-compiled implementation shown in Algorithm 11 that combines the advantages of vectorized and compiled approach. As shown in Section 3.6.4, the bucket chain following part of lookup is best to be left vectorized. In other parts, there are opportunities to apply compilation.

The first idea is to compile the full sequence of hash/rehash/bitwise-and and initial lookup into a single primitive. The hashing/rehashing/bitwise-and part re-iterate the compilation benefits of Project primitives as discussed in Section 3.2. Initial lookup requires a random access to the buckets array B. Interleaving it with more computations helps to hide the

latency of this memory access, especially when selection percentage is high.

The second step combines all operations that can be done on a single tuple in the hash table after it is accessed. When the hash table is in NSM, this would mean at most one TLB miss, compared to one miss per attribute in the vectorized implementation. Therefore, check and iterative re-check are combined. Then, if the check was successful, fetching the values into result vectors is performed at once. If not, the position of the next tuple in the bucket chain is saved in the $\vec{Pos}$ vector.

The only operation left to be done with vector granularity, is to advance to all the next tuples for which neither the check was successful and values already fetch nor the chain ended. After the loop, only hits have to be selected into the $\vec{Hits}$ vector. Code for the primitives and pseudocode of the operator level logic is shown in Algorithm 11. Since the hashing and lookup initial phases and checking, fetching and lookup next phases are joined into one and could not be measured on a finer granularity, they are shown in Figure 3.9 merged as lookup initial and checking. The split-time bar graphs of Figure 3.9 confirm that phases we compile perform better than their vectorized counterparts.

Partially compiled NSM (line "F") is the best solution, thanks to its efficient compiled multi-column checking and fetching, compiled hashing, and its vectorized independent memory accesses during chain-following.

3.7. Software engineering aspects of Vectorization and Compilation

So far in this chapter we discussed the aspects of performance of query execution routines. It was shown that combining both techniques of vectorization and JIT allows to choose the best performance. This section will discuss other aspects, connected with general system complexity and maintenance.

Figure 5.1 in [Zuk09] summarizes the comparison of a tuple-at-a-time interpreted system, column-at-a-time MonetDB and vector-at-a-time MonetDB/X100 which became VectorWise. Inspired by it, we present Table 3.1, similarly comparing a vectorized, single-loop compiled and the multi-loop compiled hybrid solution we propose. The first part of the table summarizes the performance aspects discussed in previous sections, while this section will discuss the other aspects.

3.7.1. System complexity

Having both vectorization and JIT code generation in a database system obviously requires increased system complexity. It seems however that implementing elements of one into a system based on the other could be done relatively easy, so that the complexity of implementing such a combined system is definitely less than the "sum" of complexities, as suggested with a question mark in Table 3.1.

Vectorization in a Compiled system

In a single-loop compiled system introducing vectorization boils down to adding additional materialization points in the execution algorithms. This is in opposition to what the Hyper system tries to achieve [KN10]. It can be done by introducing new code generation patterns that implement algorithms which instead of processing one tuple in a loop body accumulate a vector of input coming from the previous iterations and produce some operations as inner loops working on that accumulated vectors one operation vector-at-a-time. As described in

	Vector-at-time	Single-loop comp. tuple-at-a-time	Hybrid
Instruction independence	**very good**	poor	**very good**
SIMD opportunities	**very good**	poor	**very good**
Branch mispredictions	**can be avoided**	unavoidable	**can be avoided**
Data access pattern	sometimes poor	sometimes poor	**better**
Function calls	few	**extremely few**	**very few**
Interpretation during execution	small	**none**	**very small**
System complexity	some	some	sum?
Compiled code complexity	static, simple code	high	**medium**
Compilation time	not applicable	big	**smaller**
Compilation reusability	not applicable	hard/none	**some**
Optimization and profiling	**simple**	hard	**simple**
Error detection and reporting	**simple**	hard	medium

Table 3.1: Comparison of various aspects of a purely vectorized interpreted system, with single-loop compiled system and the proposed hybrid solution.

Section 2.4.2, the HIQUE system authors claim that adding new algorithms to code generation in a compiled system is relatively easy [KVC10].

Everything that can be done in an interpreted system can also be done in a compiled system. It is just a matter of increased complexity of code generation engine. Just like everything that can be implemented in a high level language can of course be implemented in assembly. It results in higher complexity of the compiled code. The question is when the added complexity is worth it.

Compilation in a Vectorized system

From our experience of actually using this approach, it seems that it is more work to add JIT query compilation to a vectorized system than the other way around. Completely new components have to be added to support source code generation, integration with a compiler, management of the generated machine code etc. The good thing about these components is that they are mostly independent from the existing system, so they can be designed and implemented from scratch, not dealing with any existing system limitations.

The only interconnection with existing query execution is first to analyze the query plan to seize compilation opportunities, and then to connect the generated and compiled functions into execution. This can be done easily in VectorWise, by making the JIT functions act like existing primitive functions, wrapped into the Expression execution framework. More work is required for the proper detection and generation of these function. It will be described in Chapter 5.

3.7.2. Compilation overheads

Another source of overheads in a system that uses JIT compilation is the cost of compilation itself. A system that compiles whole queries has two significant disadvantages over a hybrid approach.

- Compiling whole queries means more generated code, and therefore more compilation time required, compared to compiling only fragments.

- Furthermore, when whole queries are compiled monolithically, there is less possibility for reusing the compiled code. In a given workload, many fragments of different queries may repeat. If the system compiles the whole query, it is unable to reuse the compiled code for an other query which shares some fragments of the query plan. If fragments are compiled separately, they can be buffered and reused in a much more flexible way.

We will discuss the compilation infrastructure in details in Chapter 4, showing that our JIT compilation of only fragments of execution is very fast.

3.7.3. Profiling and runtime performance optimization

Another interesting aspect of a vectorized system is that its performance can be very easily profiled. Each primitive is a separate function operating on a vector of values. This granularity of function calls is coarse enough to allow measuring the actual performance of each primitive, including rich, detailed information like from CPU hardware counters without introducing significant measurement overhead.

This way, a vectorized system can obtain detailed and highly accurate information about the performance of each part of query execution. Such information can be used for runtime adaptation of the query plan and for providing feedback to the query optimizer. Different implementations of the same primitive can be interchanged at runtime based on their performance, e.g. the different selection implementations in Section 3.3. Changing between alternative methods can be done at vector-level granularity during execution. In long running analytical queries different approaches can be tried and measured at the beginning of query execution, before settling on the best methods. Then this choice can be periodically reevaluated.

With dynamically compiled fragments of queries there is even more room for runtime evaluation and adaptation. The generated functions may be directly compared with the primitives that they replaced.

The finer granularity of working on fragments of queries may also help other maintainability aspects, such as detecting and pinpointing errors.

The possibility of fine-grained profiling helped us in our experiments. In Figures 3.8 and 3.9 in Section 3.6 we were able to measure split performance of various phases of the algorithm of the vectorized implementations. We could not do that for the fully compiled implementation.

When operations are compiled together monolithically and performed tuple-at-a-time, they cannot be accurately profiled, because starting and stopping counters around simple operations on single tuples would be more costly than the operations itself. Some information could be derived from sampling methods, but it will not be accurate and when working on code compiled from a dynamically generated source, it is more difficult to map the profile samples in the machine code back to the corresponding operation templates from which the code was generated.

Although it is relatively easy to implement new code generation patterns and find basic bugs in them, it is really hard to find more obscure errors at runtime in the generated JIT code. Such code cannot be debugged with debuggers like GDB since they cannot link it to symbols in existing source code. Even when an error in the generated code is found, it still has to be traced back to the correct source generation pattern. Altering the pattern can result in errors in other unforeseen situations when a different dynamic combination of compiled operations occurs. Of course, there is less room for errors and they are easier to pinpoint when the compiled fragments are smaller.

Summarizing, "holistic" compilation such as proposed in HIQUE [KVC10], which generates monolithic code for whole queries misses some opportunities of flexible runtime adaptation and optimization.

3.8. Summary

For database architects seeking a way to increase the computational performance of a database engine, there might seem to be a choice between vectorizing the expression engine versus introducing expression compilation. Vectorization is a form of block-oriented processing, and if a system already has an operator API that is tuple-at-a-time, there will be many changes needed beyond expression calculation, notably in all query operators as well as in the storage layer. If high computational performance is the goal, such deep changes cannot be avoided, as we have shown that if one would keep adhering to a tuple-a-time operator API, expression compilation alone only provides marginal improvement.

Our main message is that one does not need to choose between compilation and vectorization, as we show that best results are obtained if the two are combined. As to what this combining entails, we have shown that "loop-compilation" techniques as have been proposed recently can be inferior to plain vectorization, due to better (i) SIMD alignment, (ii) ability to avoid branch mispredictions and (iii) parallel memory accesses. Thus, in such cases, compilation should better be split in multiple loops, materializing intermediate vectorized results. Also, we have signaled cases where an interpreted (but vectorized) evaluation strategy provides optimization opportunities which are very hard with compilation, like dynamic selection of a predicate evaluation method or predicate evaluation order.

Thus, a simple compilation strategy is not enough; state-of-the art algorithmic methods may use certain complex transformations of the problem at hand, sometimes require run-time adaptivity, and always benefit from careful tuning. To reach the same level of sophistication, compilation-based query engines would require significant added complexity, possibly even higher than that of interpreted engines. Also, it shows that vectorized execution, which is an evolution of the iterator model, thanks to enhancing it with compilation further evolves into an even more efficient and more flexible solution without making dramatic changes to the DBMS architecture. It obtains very good performance while maintaining clear modularization, simplified testing and easy performance- and quality-tracking, which are key properties of a software product.

From the perspective of introducing JIT compilation into the VectorWise system, the following areas turned out to be the most interesting:

- combine arithmetic calculation in projections.
- revise VectorWise hash tables implementation, introducing JIT compilation for parts of the hash table probing phase.

However, we believe that investigation of other elements of the system will reveal many additional opportunities for JIT application.

Chapter 4

JIT Compilation infrastructure

To be able to support JIT compilation, a system like VectorWise needs an infrastructure that allows it to connect to a compiler and to dynamically link the JIT generated code to itself. In this chapter we study the suitability of different compilers as a JIT compiler inside the VectorWise system. Experiments in Chapter 3 used mostly gcc, but also revealed some fundamental differences between compilers. Also, the overhead of invoking the compiler was not taken into account, only the execution time of already compiled code was measured.

First, in Section 4.1 we introduce the three considered compilers – gcc, icc and clang. In Section 4.2 we evaluate the compilers by compiling the whole VectorWise system itself. We measured both compilation time and the compiled system performance on the TPC-H benchmark. Different proof-of-concept implementations of JIT compiler are discussed and tested in Section 4.3. The proofs-of-concept were afterward turned into a JIT compilation infrastructure inside the VectorWise system.

4.1. Compilers

For our JIT compilation we want to be able to compile code generated in C, as opposed to some systems like System R [D. 81] or Hyper [KN10], which generate assembly or assembly-level low level language directly. Therefore we depend on the compiler's code generation and low level optimization abilities. This gives two important benefits:

1. Code generation complexity. It is much easier to generate source code in a high level language without the need of low level optimizations. Also, snippets of existing code from the system can be reused as templates and stubs for the code generator.

2. Portability and architecture independence. Source generated in a high level language like C is translated into hardware specific machine code by the compiler.

These benefits come at a cost of bigger compilation overhead of a general purpose compiler than would be possible by generating specialized pre-optimized code, and slightly reduced control of what actually comes out at the end of the process.

4.1.1. GNU Compiler Collection – gcc

GNU Compiler Collection (gcc) is an open-source compiler system produced by the GNU Project. It has been adopted as the standard compiler by most modern Unix-like computer operating systems and is also widely deployed as a tool in commercial, proprietary and closed source software development environments. Gcc was used as the default compiler in

the VectorWise build system, and also most of experiments in Chapter 3 were conducted using it. We used gcc 4.4.2.

4.1.2. Intel Compiler – icc

Intel C++ Compiler (icc) is a group of proprietary C and C++ compilers from Intel Corporation. As such, it is marketed as being especially suitable for generating code for Intel processors, including automatic vectorizer that can detect vectorizable loops and generate all flavours of SSE instructions. However, we observed in Section 3.2 of Chapter 3 that it sometimes makes sub-optimal decisions about vectorization in more complex situations. On the other hand it never fails to vectorize simple loops where possible.

The VectorWise build system was already configured to use icc as an alternative compiler for the system. We used icc 11.0.

4.1.3. LLVM and clang

The Low Level Virtual Machine (LLVM) [LLV], despite its name, has little to do with virtual machines, but is a collection of modules and libraries for building compilers. These libraries operate on an intermediate code representation known as the LLVM intermediate representation ("LLVM IR"). It is a target independent assembly-like language. The LLVM libraries provide a modern target-independent optimizer working on LLVM IR, along with assembly and machine code generation support for various CPUs. LLVM has been developed since 2003, making it a relatively mature set of libraries. Hyper [KN10] generates code for its JIT compilation directly in LLVM IR, therefore being able to keep the benefit of being architecture independent, while still maintaining tight control over generated code. Hyper can also use the LLVM optimizer passes for general optimizations, while implementing some specific optimizations itself.

LLVM is also suitable for creating compiler frontends for different languages. The framework is provided to help construct abstract syntax trees and code generation into LLVM IR. One such frontend is the "clang" subproject of LLVM [cla]. Developed since 2007, it is quite a young compiler, but it depends on the much more mature set of LLVM libraries for the compiler back-end. Clang claims to be significantly faster than gcc, which is true as we demonstrate in Section 4.2. One of the known current deficiencies of LLVM and clang is that it does not perform automatic SIMDization of loops optimizations yet. There is however work underway towards this functionality, the Polly project [Gro10], which will hopefully be included in the future releases. We used the currently most recent LLVM+clang release 2.9.

A strong reason to consider using LLVM and clang as a JIT compiler is their modular design as a set of libraries with relatively well documented API. This way, a JIT compiler can be constructed by linking directly with clang libraries and using its internal API. It will be discussed in detail in Section 4.3.2.

4.2. VectorWise compilation

The VectorWise build system was already configured to enable compilation using gcc and icc. Adding clang to the suite proved to be very easy, as it was designed to be a drop-in replacement for gcc, with highly compatible command line parameters format.

	debug build	optimized build
clang	145 s	245 s
gcc	87 s	490 s
icc	311 s	449 s

Table 4.1: VectorWise compilation time using different compilers. Debug build is without optimizations, while optimized build is a production build.

4.2.1. Build time

For each of the three compilers two VectorWise builds were created: a debug and an optimized build. The debug build turned off all optimizations, leaving only `-g` option for debugging symbols. Optimized builds use different optimization flags, depending on the compiler:

- gcc: `-m64 -O6 -fomit-frame-pointer -finline-functions -malign-loops=4 -malign-jumps=4 -malign-functions=4 -fexpensive-optimizations -funroll-all-loops -funroll-loops -frerun-cse-after-loop -frerun-loop-opt`
- icc: `-no-gcc -mp1 -O3 -restrict -unroll -axWPT -axSSE4.2`
- clang: `-m64 -O3 -finline-functions -funroll-loops`

Table 4.1 shows that LLVM/clang claims of being faster than gcc hold true for VectorWise compilation. While gcc is the fastest compiler in a non-optimized build, clang is able to compile with optimizations in half the time of gcc. Intel compiler is much slower when delivering a debug build, but the speed of compilation with optimizations is comparable to that of gcc.

4.2.2. TPC-H performance

Performance of the created builds was tested on a realistic workload for an analytical database – the whole set of TPC-H benchmark [Tra02] queries on a scale factor 10 database. The measured times are of "hot" runs of the queries – the database size was 10 GB, so on a machine with 12 GB of RAM all data can be buffered in the buffer pool, leading to in-memory execution. Two runs were conducted – with and without intra query parallelism.

Results of the benchmarks are shown in Table 4.2. In both runs, the gcc optimized build was the fastest. The results were however very close to each other. The clang build is slower by 5% in the single threaded run. However, some known deficiencies of clang, like automatic SSE generation are currently being worked on. In the multi-threaded run all compilers are within 1.2% of each other.

The difference between a SIMDizing and non-SIMDizing is smaller than it could be expected, as one of the first things that come to mind as a benefit of *vectorized* (in the meaning of operating on vectors of data) execution is the possibility to *vectorize* (in the meaning of using SIMD machine instructions) the code. However, results show that it does not really have such high impact. The reason for that is that actually the easily SSE-able operations, like map calculations and computing hash functions do not take up much of the execution time. Microbenchmarks in Chapter 3 displayed results of a few cycles per tuple in the project case study in Section 3.2 which involved SIMD opportunities, compared to hundreds of cycles in hash table operations involving random memory accesses in Section 3.6. Another reason is that even in those simple map primitives, the compilers claiming auto-vectorization are

	Single thread			Eight threads		
Query	clang	gcc	icc	clang	gcc	icc
1	2.19 s	1.96 s	1.95 s	0.56 s	0.53 s	0.54 s
2	0.24 s	0.22 s	0.20 s	0.18 s	0.18 s	0.17 s
3	0.18 s	0.17 s	0.17 s	0.17 s	0.17 s	0.17 s
4	0.16 s	0.17 s	0.18 s	0.12 s	0.11 s	0.11 s
5	0.57 s	0.57 s	0.57 s	0.49 s	0.49 s	0.50 s
6	0.15 s	0.15 s	0.23 s	0.10 s	0.10 s	0.11 s
7	0.58 s	0.61 s	0.58 s	0.29 s	0.28 s	0.28 s
8	0.65 s	0.66 s	0.63 s	0.37 s	0.39 s	0.41 s
9	4.11 s	3.69 s	3.93 s	1.48 s	1.46 s	1.44 s
10	0.77 s	0.76 s	0.76 s	0.46 s	0.46 s	0.47 s
11	0.25 s	0.24 s	0.24 s	0.17 s	0.18 s	0.18 s
12	0.49 s	0.50 s	0.51 s	0.20 s	0.21 s	0.21 s
13	3.17 s	3.06 s	3.02 s	1.18 s	1.16 s	1.15 s
14	0.35 s	0.33 s	0.35 s	0.23 s	0.22 s	0.23 s
15	0.16 s	0.16 s	0.18 s	0.14 s	0.15 s	0.14 s
16	0.92 s	0.91 s	1.01 s	0.48 s	0.47 s	0.49 s
17	0.56 s	0.56 s	0.54 s	0.26 s	0.24 s	0.25 s
18	1.93 s	1.65 s	1.80 s	0.73 s	0.69 s	0.72 s
19	1.35 s	1.38 s	1.34 s	0.40 s	0.42 s	0.40 s
20	1.75 s	1.73 s	1.69 s	0.67 s	0.67 s	0.65 s
21	2.16 s	2.21 s	2.44 s	0.88 s	0.87 s	0.90 s
22	0.76 s	0.69 s	0.65 s	0.38 s	0.37 s	0.37 s
Total time:	23.45 s	22.38 s	22.97 s	9.94 s	9.82 s	9.89 s
Power:	54077.8	55561.2	53882.7	103715.9	104527.5	103669.0
Relative:	(104.8%)	(BEST)	(102.6%)	(101.2%)	(BEST)	(100.7%)

Table 4.2: Performance of VectorWise compiled by different compilers on the TPC-H benchmark scale factor 10 on a 2.67 Ghz Intel Nehalem 920 with 4 physical (8 virtual) cores. Because total time is influenced more by the differences in long running queries, an alternative metric called "Power" is also presented, which is proportional to the geometric mean of query times and is more objective.

often not able to SIMDize the loops correctly. Closer investigation of generated assembly for individual primitives show, that both gcc and icc do not generate SSE code for some of trivially SIMD-izable loops in VectorWise primitives. The pattern of whether they succeed or fail is seemingly random. VectorWise build system could be improved with automatic means of inspecting and reporting the quality of compiled primitives, so that such imperfections would be automatically detected and corrected.

4.3. Implementation

In this section we discuss the objective of building a generic JIT compiler infrastructure, with an API capable of compiling arbitrary code provided as strings in memory and extracting pointers to functions given their names.

Algorithm 12 A short snippet demonstrating how a simple JIT compiler is able to compile arbitrary code from a string by using an external compiler and dynamically linking to the created library can be. Actual system provides additional functionalities omitted in this snippet, like system calls error checking, compilation errors checking, platform independence, more advanced ability to track used memory and to free the no longer needed libraries, or to use different compilers with different command line options to the calls. Still, a full implementation fits in under 200 lines of code.

```
/* JIT compiles the source from string into a library and loads it */
void *compile_lib(char *src) {
    // Write generated source to temporary file.
    FILE *f = fopen("/tmp/jitsrc.c", "w");
    fprintf(f, "%s", src);
    fclose(f);
    // Prepare target library filename.
    char libname[15];
    strcpy(libname, "/tmp/libXXXXXX");
    mktemp(libname);

    int pid = fork();
    if(!pid) { // child calls the compiler
        execl("/usr/bin/gcc", "gcc", "-O3", "src.c",
              "-fPIC", "-shared", "-o", libname, NULL);
    }
    // parent loads the library.
    wait(pid); // wait for compiler to finish.
    unlink("/tmp/jitsrc.c");
    return dlopen(libname, RTLD_NOW | RTLD_GLOBAL);
}

/* gets a function from the JIT compiled library */
void *get_function(void *libhandle, char *func) {
    return dlsym(libhandle, func);
}

/* free a no longer needed library */
void free_lib(void *libhandle) {
    dlclose(libhandle);
}
```

4.3.1. External compiler and dynamic linking

Most other database systems featuring JIT compilation write the generated source to a file and use the compiler externally on it. It can be done by forking a new process which invokes a C compiler, and after waiting for it to finish, dynamically linking to the created library, as is done in the HIQUE system [KVC10]. The JAMDB system [RPML06] is written in Java, so it can use built-in JIT capabilities of Java, simply writing a class into a file, and using Java ClassLoader to load it. AT&T's Daytona system [Gre99] compiles a standalone program specialized in executing one query (possibly with wildcard parameters) in an architecture without a database server, and the client runs it directly.

Algorithm 12 demonstrates the simplicity of the external approach. To interface with the compiler the generated source has to be written to a disk file, but as the compiler will be accessing it directly afterward, it will most probably be able to read it from disk caches in memory instead of actually using the hard drive. If the source file is removed afterward, it is possible that it will even actually never be synced to disk. The Linux functions `fork` and `exec` family, `dlopen`, `dlsym`, `dlclose` have Windows counterparts `spawn`, `LoadLibrary`, `GetProcAddress` and `FreeLibrary`, enabling to port this approach to the Windows version of the system. This approach is compiler independent. Any compiler can be used, if provided

with correct path and command line options.

4.3.2. Using LLVM and clang internally

The modular design of LLVM and clang, with well defined API makes it possible to use it internally, linking directly to LLVM libraries. The LLVM suite features a library designed explicitly for JIT compilation, the ExecutionEngine [LLV].

We used two presentations from LLVM Developers' Meetings as an example of how to implement such a JIT compiler using LLVM. The first one [Beg08] discussed the LLVM JIT API in general. The second one discussed Cling [Nau10], a prompt-based C interpreter developed at CERN, extending the language to enable interactive writing and executing C-like code as in e.g. Python, Ocaml, R etc. We also based our implementation on the "clang-interpreter" example in the clang code repository.

The goal was to make the interaction with LLVM objects as simple as possible, while saving the overheads:

- do not create external processes,
- have a "warm" pre-initialized compiler ready to accept work,
- read source directly from memory and generate executable code also directly into memory,

The source code for achieving these goals is longer than that for the external approach, and would not fit in a single snippet, but conceptually it is simple. On a high level, the following steps have to be performed:

1. The input location has to be overridden to read source from memory instead of a file.
2. A clang instance has to be invoked to parse the source and generate a Module with LLVM IR, and apply LLVM optimization passes to it.
3. The resulting Module can be put into JIT ExecutionEngine.
4. ExecutionEngine can be asked to generate machine code for given symbols from the Module and return function pointers.

Both the LLVM ExecutionEngine object and clang CompilerInstance object, together with a number of helper objects, can be created once and reside ready waiting for incoming compilations.

4.3.3. JIT compilation infrastructure API

We abstracted the JIT compiler from other layers of the system by using an uniform API with which all the compilers have to comply.

A global function `jit_exclusive_get` is used by a `JitContext` (to be described in Section 5.3.2) to request its need to use a compiler (and afterward to release it using `jit_release`). A handle is returned, with pointers to functions of the API. Such a handle global request method is also used to control concurrency problems with multiple compilations performed at once. For example, the LLVM libraries and clang are not adapted to be used in a multi threaded environment, with multiple compilations going on at the same moment, so locking has to be done when using the internal LLVM implementation of the JIT compiler.

The decision on the strategy of which compiler to choose has not been determined. We could be using always one compiler, or switching between different ones for different queries. Different paths and command line arguments to externally called compilers can also be used. These options could be exposed as VectorWise configuration options to the database user. This would enable the user to specify any compiler, any path and any set of arguments to the C compiler. In the scope of this thesis we implemented a simple code switch that let us test different JIT compilers.

The API of a concrete compiler is actually very simple and consists of only three functions:

- `void* jit_compile(char* source)` compiles and loads the C code saved in a string. The result is a handle to the loaded library.
- `void* jit_get_function(void* library, char* symbol)` retrieves a function with the given symbol name from the previously compiled library.
- A global method `jit_clear` to unload all JIT compiled code.

In case of an externally used compiler, the `jit_compile` method differs only by using different paths and command line line arguments to different compilers. The `jit_get_function` method is the same, since it just deals with libraries dynamically loaded with `dlopen`, which are the same, no matter what compiler was used. The internal clang compiler implementation uses the LLVM infrastructure and therefore the implementation of `jit_compile` and `jit_get_function` does not have any similarities to the external one. The `jit_clear` method for LLVM compiler clears the ExecutionEngine object. For external compilers it closes all the loaded libraries with `dlclose`.

It would be useful to have a method for controlling the memory usage and unloading JIT primitives more selectively. There are however a few problems with that:

- The ExecutionEngine object in LLVM is able to report the amount of memory occupied by compiled code, and we can even replace the JITMemoryManager with our own implementation. It is not so easy however with the dynamically loaded libraries in the external compiler implementation. There is no portable way to ask how much memory is a library loaded with `dlopen` actually taking up. A possible estimate for this would be to check the size of the library `.so` file. On Linux the `/proc/PID/maps` file could be queried to look where the library is mapped into the process memory.
- We compile multiple functions in a single compilation to save overhead. Again, while LLVM has a method to free machine code for just a single symbol, with `dlclose` we can only unload whole libraries. A hypothetical memory manager would have to maintain relations between functions coming from the same library and only unload the library after unloading of all symbols have been requested.

Therefore, a more advanced implementation of JIT code memory management and an API for that remains future work.

4.3.4. Evaluation

A proof-of-concept implementation was written as a microbenchmark of the described solutions. We compared compilation speed of the external approach using all three compilers and using clang and LLVM internally, both in not-optimized (-O0) and optimized (-O3) compilation.

Execution time of `jit_compile` and subsequent calls to `jit_get_function` for all functions in the compiled code was measured. The benchmark was performed on three source files:

Algorithm 13 Queries 1 and 19 of the TPC-H benchmark. Version of our code generator we used when evaluating different JIT compilers was able to recognize and generate code of primitives for the projection calculations like `l_extendedprice*(1-l_discount)*(1+l_tax)` in Query 1, and for complex selection conditions in Query 19, together with necessary declarations producing 10 kB and 34 kB of C source code to be JIT compiled.

```
# Query 1
select
        l_returnflag, l_linestatus,
        sum(l_quantity) as sum_qty,
        sum(l_extendedprice) as sum_base_price,
        sum(l_extendedprice*(1-l_discount)) as sum_disc_price,
        sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge,
        avg(l_quantity) as avg_qty,
        avg(l_extendedprice) as avg_price,
        avg(l_discount) as avg_disc,
        count(*) as count_order
from lineitem
where l_shipdate <= date '1998-12-01' - interval '90' day
group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus;

# Query 19
select sum(l_extendedprice * (1 - l_discount) ) as revenue
from lineitem, part
where
        (
                p_partkey = l_partkey
                and p_brand = '[BRAND1]'
                and p_container in ( 'SM_CASE', 'SM_BOX', 'SM_PACK', 'SM_PKG')
                and l_quantity >= [QUANTITY1] and l_quantity <= [QUANTITY1] + 10
                and p_size between 1 and 5
                and l_shipmode in ('AIR', 'AIR_REG')
                and l_shipinstruct = 'DELIVER_IN_PERSON'
        )
        or
        (
                p_partkey = l_partkey
                and p_brand = '[BRAND2]'
                and p_container in ('MED_BAG', 'MED_BOX', 'MED_PKG', 'MED_PACK')
                and l_quantity >= [QUANTITY2] and l_quantity <= [QUANTITY2] + 10
                and p_size between 1 and 10
                and l_shipmode in ('AIR', 'AIR_REG')
                and l_shipinstruct = 'DELIVER_IN_PERSON'
        )
        or
                (
                p_partkey = l_partkey
                and p_brand = '[BRAND3]'
                and p_container in ( 'LG_CASE', 'LG_BOX', 'LG_PACK', 'LG_PKG')
                and l_quantity >= [QUANTITY3] and l_quantity <= [QUANTITY3] + 10
                and p_size between 1 and 15
                and l_shipmode in ('AIR', 'AIR_REG')
                and l_shipinstruct = 'DELIVER_IN_PERSON'
        );
```

- Hello World – a simple source with one function printing "Hello World!". This tests the plain overhead of starting the compiler, loading the library, linking and resolving symbol references (a reference to standard library function `printf`).

- Query 1 of TPC-H. The queries are presented in Algorithm 13. The JIT compiled source is not a program for evaluating the whole query, but a small portion of it that an early version of our source generator was able to produce. It compiles JIT primi-

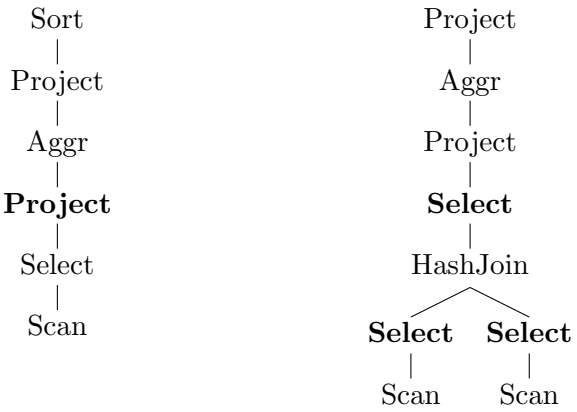


Figure 4.1: Query plans of Query 1 and Query 19 of TPC-H. The operators for expressions in which source code was used in the benchmark are written in bold.

Test case / JIT compiler		gcc	icc	clang	clang-internal
Hello World	-O0	23.4 ms	139.7 ms	25.8 ms	0.9 ms
	-O3	24.9 ms	145.3 ms	27.4 ms	1.5 ms
Query 1 (10 kB)	-O0	38.3 ms	150.2 ms	38.0 ms	15.9 ms
	-O3	56.6 ms	193.9 ms	44.0 ms	16.9 ms
Query 19 (34 kB)	-O0	107.5 ms	205.7 ms	95.4 ms	104.8 ms
	-O3, no vec.	757.4 ms	855.6 ms		
	-O3	754.8 ms	888.9 ms	338.2 ms	301.4 ms

Table 4.3: Compilation speed of three test cases using different JIT compilers. The fast startup advantage of clang-internal is quickly diminished as size and complexity of JIT compiled source increases. Clang once again confirms its fast optimization capabilities. Icc has the slowest startup of all compilers.

tives for calculating the Project expressions like `l_extendedprice * (1-l_discount) * (1+l_tax)` from the query plan in Figure 4.1. They amount to 10 kB of C source code, which also includes a preamble with all declarations and type definitions necessary for using this code in VectorWise after JIT compilation. The code itself is quite bloated, as the C code generator produces each operation in a new statement, declaring and using a new local variable, similar to a Static Single Assignment (SSA) form. Therefore, the code size gets quite big, even for those few simple operations [ASU86].

- Query 19 of TPC-H. The query in Algorithm 13 has a very complicated selection condition on two tables. In query plan in Figure 4.1, this selection gets separated into three Select operators – two below a join, for conditions from only one table, and one after the join. In each of these Selects code is generated to evaluate the complex condition, producing 34 kB of source code in total.

Results are shown in Table 4.3. Internal clang implementation offers very fast startup latency, being able to deliver a compiled "Hello World" function in under a millisecond, 25x faster than calling gcc or clang externally, and over 150x faster than icc. Icc has particularly slow startup time, but catches on with gcc's speed once bigger pieces of code are compiled.

However, this advantage of internal clang implementation in fast startup time becomes less significant as the source becomes more complicated, when the compilation itself becomes

the dominating part. Nevertheless, while without optimizations clang is on par with gcc, it beats the other compilers by over a factor of 2 when optimizations are enabled, while producing code of comparable quality as was proved in Section 4.2. This doesn't account for clang not performing auto-vectorization optimizations so, to be absolutely fair, we reran the last test on gcc with `-fno-tree-vectorize` and icc with `-no-vec` flags to explicitly disable these optimizations. That however did not change anything on gcc and took less than 4% off icc's time.

The conclusion is that if bigger pieces of generated code are to be compiled at once, there is no advantage in linking directly with the compiler and using its internal API. The external approach saves complication, library dependencies and having to maintain changes to the JIT compiler each time a new version of the libraries change the internal APIs. It also has more flexibility, as any compiler can be chosen to be called. If however a model in which we often (meaning: multiple times per query) compile small pieces of code was chosen, the internal compiler would be the best choice.

4.4. Summary

Comparison of VectorWise performance when compiled by different compilers have shown that the differences are very small. On the other hand, the difference between how the compilers perform themselves is huge, with clang being over twice as fast as any other compiler. As compilation overhead is very important in JIT compilation, clang should be the first choice.

Linking with LLVM and clang and using them internally promised a low latency always-ready JIT compiler, but that turned out not to be such a big advantage, as the real-situation generated code is complex enough that compiler startup latency is not the dominating factor. Moreover, maintaining the dependency to LLVM libraries binds the system not only to one compiler, but also to its specific version, as minor changes in the API prevent from just linking against a different release of LLVM. Using the compiler as an external program allows full flexibility of configuring what compiler to use.

The surprising conclusion is to use the simplest solution because it works best. Using less known clang rather than gcc may result in reduced overheads. The door always remains open for any different configuration.

Chapter 5

JIT primitive generation and management

This chapter contains practical details of how the detection of JIT opportunities, source code generation and management of the compiled code was implemented in VectorWise.

Chapter 4 already described the integration with a C compiler, so we skip this part of infrastructure in this chapter. In Section 5.1 we explain how the templates of operations for which we perform code generation are extracted from the VectorWise own source code and maintained. In Section 5.2 we describe how we find opportunities for JIT compilation in the query execution tree during the Rewriter phase. Section 5.3 discusses source code generation in more detail. We explain how the object code of compiled primitives is stored and maintained for future reuse in Section 5.4.

Not all of the ideas from Chapter 3 were implemented into VectorWise in the scope of this thesis. Some future enhancements will be described in Chapter 7.

5.1. Source templates

In Section 2.2.2 we introduced primitives, which are the *working horse* functions during query execution. In JIT query compilation we would like to compile many such primitives together into a single function. It would be good to be able to reuse parts of source code of the original primitives when generating JIT source for new ones.

5.1.1. Primitive templates

Usually a primitive is a little more complicated than was shown in Algorithm 1 on page 18. In addition to the operation performed in the loop, there is often some initialization prologue before the loop and an epilogue after. These are used for checking and returning execution errors such as overflows, division by zero etc. There are therefore three things that we would like to extract from the source of a primitive for code generation of its dynamic counterpart: the prologue, the in-loop operation itself and the epilogue.

Help comes from the fact that primitives are generated using macros. The macro language Mx (developed internally at CWI for the MonetDB database [Bon02]) and the C preprocessor are used to expand combinations of a primitive operation in different dimensions: types of arguments and result, working on vectors or scalars, with or without selection vector. A Mx macro is used to expand the different combinations and provide a template for the main primitive loop, while the operation itself is wrapped into a C preprocessor function.

Algorithm 14 A (simplified) example of how Mx and C preprocessor macro languages are used to expand primitive functions. The `impl_` and `decl_` macros are used to generate implementations of all primitives and an array of with pointers to all of them.

```
/* Operations defined as preprocessor macro */
#define sint_div(res,a,b) \
    { int div_by_zero=((b)==0); res=(a)/((b)+div_by_zero); err |= div_by_zero; }
#define sint_div_ret
    if(err) return x100_err(e, VWLOG_FACILITY, X100_ERR_INTDIVBYZERO);

/* Mx macro */
@= decl_mapop_colcol
    { "@4", "map_@1_@2_col_@3_col", (Primitive) map_@1_@2_col_@3_col, },
@= impl_mapop_colcol
uidx map_@1_@2_col_@3_col
        (uidx n, @4*__r res, @2*__r col1, @3*__r col2, uidx*__r sel, Expr *e) {
    ulng err = 0;  /* all primitives use a ulng err variable */

    #ifdef @1_prologue
    @1_prologue  /* would expand a sint_div_prologue macro if it existed */
    #endif

    /* Application of a macro that would create the "for" loop
       for sel. vec. and no sel. vec. */
    @:impl1(@1(res[i],col1[i],col2[i]))@

    #ifdef @1_ret
    @1_ret  /* @1 will expand to sint_div.
               C preprocessor will expand sint_div_ret macro */
    #else
    generic_ret /* Generic epilogue macro */
    #endif
    return n;
}
@= implsel1 { uidx i=0,j=0; for(j=0; j<n; j++) { i = sel[j]; @1; } }
@= implnosel1 { uidx i=0; for(i=0; i<n; i++) { @1; } }
@= impl1 { if (sel) { @:implsel1(@1)@ } else { @:implnosel1(@1)@ } }

/* Application of the macro to create a sint_div primitive */
@:decl_mapop_col_col(sint_div,sint,sint)@
@:impl_mapop_col_col(sint_div,sint,sint)@
```

There are few top-level Mx macros that accept an operation name as one of their arguments. If the operation name is `OP` (example: `sint_div`), then it would expect that there is a C preprocessor macro defined for this operation `OP` and optionally a macro for the prologue `OP_prologue`, and epilogue `OP_ret`. Algorithm 14 shows an example how Mx and C preprocessor are used to expand primitive functions. Mx uses the `@=` keyword for macro definition, `@:` for call and `@1`, `@2`, ... for macro arguments expansion. The snippet is simplified, in fact more levels of macro expansion would be used to avoid code duplication, there will be more expansion for vector and scalar attributes etc. The important fact here is that just one macro will be called for a very broad class of operations, such as all binary map operations. It will be passed the operation itself as one of the parameters (here: `@1` as `sint_div`, further expanded by C preprocessor). The `impl_` macros expand to primitives implementations. The `decl_` macros create an array of definitions, which is later turned into a hash table used to lookup primitives by name.

Therefore, intercepting the operations and saving them as strings shall require modifying just a few of the top-level Mx macros. We take advantage of the fact that C preprocessor is able to turn and properly escape the argument of a macro into a string using `#define __QUOTE(x) #x`, and use it to save `OP`, `OP_prologue` and `OP_ret` to global strings in the `impl_`

Algorithm 15 Extracting source templates from primitives using macro tricks.

```
#define _QUOTE(x) #x

@= impl_FOO
...
#define err $0
const char* _@1_@2_@3 = _QUOTE(@1($1,$2,$3));
const char* _@1_@2_@3_prologue =
    #ifdef @2_prologue
    _QUOTE(@1_prologue;);
    #else
    "";
    #endif
const char* _@1_@2_@3_ret =
    #ifdef @1_ret
    _QUOTE(@1_ret);
    #else
    _QUOTE(generic_ret);
    #endif
#undef err

@= declsrc_op_colval
    { "@5", "_@1_@2_@3", 2, &_@1_@2_@3, &_@1_@2_@3_prologue, &_@1_@2_@3_ret },
```

macro. In these strings we use `$0`, `$1`, ... to mark placeholders for the error variable (`$0`, taking advantage of the fact, that all primitives in the system use `err` as the error variable name), result (`$1`) and operands. Later, when generating source, they will be substituted with correct variable names.

An array with declarations of the operations saved as strings can be created in the same way as an array of primitives is created using another series of macros, `declsrc_`, which are be expanded like `impl_` and `decl_`. The resulting source templates are then turned into a hash table, similar to that with primitives, which can be used to lookup if for a given primitive we also have an extracted template.

A code snippet showing how to force saving operations source code into strings in parallel with implementing them in the system is shown in Algorithm 15.

Templates for code generation can be extracted from the existing primitives source code with small modifications in only a few places of macro expansion. There are some exceptions, and some things have to be done in a more complicated way, but in general the implementation is clean and not invasive.

There is a system table in the VectorWise system that can be queried to view all available source templates. It can be done using:

```
SysScan('jit_templates', ['name', 'return_type', 'src', 'prologue', 'ret']);
```

5.1.2. Source preamble

There are some global definitions and declarations that are needed and used by the primitives and therefore have to also be included in the generated code of every compilation. We achieve this by creating a preamble that contains all necessary definitions, which is them pasted at the beginning of each generated compilation unit.

It contains type definitions of different types (uidx, ulng, slng, uint, sint etc.) compatible with the system platform and configuration, enums for status and error reporting, some structs and function declarations. It is all forwarded into a global string using some more C preprocessor tricks.

5.2. JIT opportunities detection and injection

We implemented the detection of possibilities for dynamic compilation and transformation of the query plan in the Rewriter phase, which was briefly introduced in Section 2.2.1 on page 16. For tree manipulations the Rewriter uses Tom [Tom] libraries, an extension language that can be used together with C, designed to support manipulations on tree structures.

We implemented three additional Rewriter rules, applied at the end of the Rewriter phase. Each of them traverses and transforms the `xnode` tree, as was described in Section 2.2.1. An example of `xnode` tree with additional information annotated by the Rewriter is shown in Figure 5.1.

5.2.1. Fragments tagging pass

The first pass traverses the tree and tags fragments that we are able to generate code for and compile. In the previous Rewriter passes, the nodes were annotated with a hint about what primitive will be used for the operation in that node. As described in Section 5.1.1 the source templates extracted from primitives are named using a scheme corresponding to the names of primitives and put into a hash table. We can therefore lookup if we have a source template for the primitive annotated to a given node, and if so, annotate the node with a hint about this template. Depending on the context we can also tag the node with a hint about what kind of operation will be required – a selection primitive, a map primitive etc.

5.2.2. Fragments generation and collapsing pass

In the second pass we traverse the tree again to find fragments that we will actually compile. In the scope of this thesis we implemented a naive rule, where we always compile the biggest possible fragments.

Therefore, this rule finds strongly connected components of the tree that have been tagged by the previous pass. For each strongly connected component it creates a tree of new structures called `snode`s, which contain all information necessary for code generation. It collapses the fragment into a single `xnode`, rearranging the children of all leaves of the fragment to be children of this `xnode`. It might have happened that the subtree used the same input parameter multiple times (e.g. many conditions concerning the same attribute). In such a situation, the duplicates would be detected, and the aliased inputs would be collapsed into one. The `snode` tree is attached to the `xnode` for future code generation.

It may also be detected that such a fragment of the tree was already compiled in some previous query, and that the resulting JIT primitive may be reused. In this case, the buffered JIT primitive will be retrieved and attached to the `xnode` directly instead of the `snode`. Another possibility is that an identical fragment is present in a different part of the tree for the same query. It will also be detected, and the same `snode` will be attached rather than generating the same code twice. More about maintaining and reusing the generated primitives will be discussed in Section 5.4.

5.2.3. Compilation pass

The last pass performs compilation and attaches the compiled functions as primitives to the `xnode`s. Compilation is performed only once per query – all code is generated into one string buffer, with the preamble in the front. This way the compilation overhead is minimized, since the compiler only needs to be called once.

```
<AGGR>:  props: $$usedcols(flag sumprice) columns(flag sumprice)
|_<PROJECT>:  props: $$usedcols(flag resprice) columns(resprice flag)
| |_<MSCAN>:  props: $$usedcols(_lineitem._l_extendedprice _lineitem._l_tax _lineitem._l_returnflag)
| | |               columns(_lineitem._l_extendedprice _lineitem._l_tax _lineitem._l_returnflag)
| | |_<ASSIGNMENT>: _lineitem
| | | \_<STRING>: str('_lineitem')
| | \_<LIST>: COLGEN(Raw) props: $varnr(0)
| |   |_<STRING>: str('_l_extendedprice') props: isconst(0) dmax(1.0495e+07) dmin(90091)
| |   |                                          type(sint) ltype(decimal(8,2))
| |   |_<STRING>: str('_l_tax') props: isconst(0) dmax(8) dmin(0) type(schr) ltype(decimal(2,2))
| |   \_<STRING>: str('_l_returnflag') props: isconst(0) dmax(82) dmin(65) type(sint) ltype(char(1))
| \_<LIST>: COLDEF(Project)
|   |_<ASSIGNMENT>: resprice props: type(sint) isconst(0) dmax(1.13345e+09) dmin(9.0091e+06)
|   |                               ltype(decimal(21,4)) $varnr(0)
|   | \_<OPERATOR>: str('*') props: $primitive(map_*_sint_col_sint_col) type(sint) isconst(0)
|   |   |                           dmax(1.13345e+09) dmin(9.0091e+06) ltype(decimal(21,4)) $varnr(0)
|   |   |_<IDENT>: _lineitem._l_extendedprice props: isconst(0) dmax(1.0495e+07) dmin(90091)
|   |   |                                              type(sint) ltype(decimal(8,2))
|   |   \_<CALL>: str('sint') props: $primitive(map_sint_schr_col) type(sint) dmax(108) dmin(100)
|   |     |                        ltype(decimal(13,2)) isconst(0) $varnr(0)
|   |     \_<OPERATOR>: str('+') props: $primitive(map_+_schr_val_schr_col) type(schr) isconst(0)
|   |       |                           dmax(108) dmin(100) ltype(decimal(13,2)) $varnr(0)
|   |       |_<OPERATOR>: str('*') props: $primitive(map_*_schr_val_schr_val) type(schr) isconst(1)
|   |       | |                          dmax(100) dmin(100) ltype(decimal(13,2)) $varnr(0) constval(100)
|   |       | |_<CALL>: str('schr') props: $primitive(map_schr_str_val) type(schr) dmax(1) dmin(1)
|   |       | | |                          isconst(1) ltype(sint) $varnr(0) constval(1)
|   |       | | \_<STRING>: str('1') props: dmax(1) dmin(1) type(str) ltype(varchar(1))
|   |       | |                             isconst(1) constval(1)
|   |       | \_<CALL>: str('schr') props: dmax(100) dmin(100) type(schr) ltype(slng) isconst(1)
|   |       |   |                          $primitive(map_schr_str_val) $varnr(0) constval(100)
|   |       |   \_<STRING>: str('100') props: dmax(100) dmin(100) isconst(1) type(str) ltype(str) $varnr(0)
|   |       \_<IDENT>: _lineitem._l_tax props: isconst(0) dmax(8) dmin(0) type(schr) ltype(decimal(2,2))
|   \_<ASSIGNMENT>: flag props: isconst(0) dmax(82) dmin(65) type(sint) ltype(char(1))
|     \_<IDENT>: _lineitem._l_returnflag props: isconst(0) dmax(82) dmin(65) type(sint) ltype(char(1))
|_<LIST>:
| \_<IDENT>: flag props: isconst(0) dmax(82) dmin(65) type(sint) ltype(char(1))
|_<LIST>: COLDEF(Aggr)
  \_<ASSIGNMENT>: sumprice props: type(s128) isconst(0) dmax(4.98499e+21) dmin(0) ltype(decimal(22,4))
    \_<CALL>: str('sum128') props: type(s128) isconst(0) dmax(4.98499e+21) dmin(0) ltype(decimal(22,4))
      \_<IDENT>: resprice props: type(sint) isconst(0) dmax(1.13345e+09) dmin(9.0091e+06)
                                 ltype(decimal(21,4)) $varnr(0)
```

Figure 5.1: An `xnode` tree with attached properties, pretty-printed from the system log.

The pass through the tree will find all `xnode`s with attached `snode`s and find for them the resulting compiled JIT primitive, which will be attached to them just as normal primitives were attached to `xnode`s previously. Since they work in the same way, they can be put into execution with almost no changes in the Builder and later phases.

5.2.4. Limitations

The Rewriter was chosen as the phase in which to implement JIT opportunities detection and code generation because operating on the `xnode` tree with new pluggable rules was the least intrusive for the system and easiest to do. However, the `xnode` tree in the Rewriter phase corresponds only to the parsed VectorWise algebra, and as such does not contain all the information about query execution.

In Figure 5.1 we can see that the Rewriter semantic analysis rules attach a lot of meta information to the `xnode`s: value ranges, types and the primitives that will be used. However, the `Aggr` node has only three children: its input (the `Project`), a list of "group by" keys and a list of aggregate functions. From that we cannot see e.g. what form of `Aggr` will

```
[SNODE_MAPPRIM]:  res_type: sint, res_var: r0, err_var: e1, const: 0 value: ptr(7f921c03a770)
\_[SNODE_MAPOP]:  res_type: sint, res_var: r2, err_var: e3, const: 0 value: prim(_*_sint_sint)
  |_[SNODE_INPUT]:  res_type: sint, res_var: r4, err_var: e5, const: 0 value: int(0)
  \_[SNODE_MAPOP]:  res_type: sint, res_var: r6, err_var: e7, const: 0 value: prim(_sint_schr)
    \_[SNODE_MAPOP]:  res_type: schr, res_var: r8, err_var: e9, const: 0 value: prim(_+_schr_schr)
      |_[SNODE_CONST]:  res_type: schr, res_var: r10, err_var: e11, const: 1 value: str(100)
      \_[SNODE_INPUT]:  res_type: schr, res_var: r12, err_var: e13, const: 0 value: int(1)
```

Figure 5.2: An `snode` tree used to generate code for a `*(l_extendedprice,+(1,l_tax))` expression. The `snode`s contain information such as return type and variable names. Depending on their type, they contain additional specialized information. It can be a pointer to correct primitive template, index of an input parameter or for constants it is their constant value. The pointer in the root node value is a shortcut to an array of all input parameters.

be used. There are no `xnode`s corresponding to operations such as computing hashes, hash table lookups and inserts, or shifting the bits of the key values to create a direct aggregation index. Direct aggregation is an optimized aggregation for small domains, where the keys are used directly as indexes, without a hash table. The aggregate functions in the tree are not connected in any way to operations calculating the group positions to which the given value should be added. All these operations are created and resolved in the Builder phase rather than Rewriter. It is the logic of the Aggregation operator allocator at operator build time that will decide whether to use hash or direct aggregation based on the information about ranges of the key attributes. It will create expressions that calculate the group positions and connect them as inputs to the aggregate functions.

Therefore, as it was already mentioned in Section 2.2.2, the final execution tree cannot be derived from the `xnode` tree in the Rewriter phase. JIT detection and generation in the Rewriter phase is sufficient for compiling together arithmetic operations in projections or selection conditions, but for more complicated operations, the logic would need to be moved to the Builder phase. There, each operator allocator would be responsible for generating operations specific to that operator. Unarguably, that implementation would be much more intrusive to the system, as pieces of JIT generation and compilation would need to be scattered in each of the affected operators.

5.3. Source code generation

In this section we will describe in more detail how the JIT primitives source code is generated from the `snode` trees.

5.3.1. snode trees

The tree of `snode`s is a syntax tree for the to-be-generated code. A `snode` is created not only for the operations, but also for constants, function inputs, function root nodes. Each `snode` has a type, and depending on the type, additional information necessary to generate code for the operation it represents:

- All nodes have an unique name for the local variable in which to output its operation result. It can be the local variable in which to write the result of some computation, or the variable in which to save the currently processed value from an input vector.
- All nodes hold the result type of the operation they represent.

- All nodes have an unique name for the local variable in which to store an error flag. It is a substitute for the `err` variable from the original primitives. Since now there are multiple operations in one primitive, we need to have multiple error flags, one for each of them. Not all operations use it, but it is impossible to tell using the preprocessor macros when extracting the source templates.
- Nodes representing primitive operations hold a reference to the proper source template.
- Nodes representing input parameters hold the name of the parameter they have to read from.
- Nodes representing constants hold the constant value they contain.

An `snode` tree for an example expression `*(l_extendedprice,+(1,l_tax))` is shown in Figure 5.2.

5.3.2. JitContext

As mentioned, there is one compilation performed per query. This compilation is managed by a `JitContext` object.

`JitContext` provides compilation-global utilities such as unique name generation. It holds all the roots of the `snode` trees for which code will be generated. In later phase, it is the `JitContext` that interfaces with the compilation infrastructure and commands to perform compilation on the string buffer it managed. In the end, it holds the compiled module object code and is queried to return pointers to the compiled functions for the `xnode`s.

`JitContext` also contains the string buffer into which the source code is generated. The string buffer is a simple utility, which is starts with the source preamble with various definitions. It provides basic functionality of printing and placeholder filling, necessary for source generation.

5.3.3. Code generation

The code is generated by walking over the tree of `snode`s and writing to the string buffer in the `JitContext`. A usual syntax tree corresponds directly to the code, and a compiler pretty-printer needs only to walk it once during generation. The `snode` tree however has to be walked multiple times in order to create the JIT primitive function:

- The first walk organizes input parameters. When there are many operations combined, some might have had the same input parameters (e.g. selection conditions on the same attribute, complex arithmetic calculations involving some attribute in more than one place). We are able to detect and combine such aliased inputs.
- The second walk of the tree generates the operation prologues before the loop.
- Two walks of the tree are done for generating code of the loop bodies – once with a selection vector, and once without.
- A fifth walk of the tree generates the operation epilogues (error checking) after the loops.

The walks are post-order, so code for the children operations is generated before the parent.

Algorithm 16 JIT primitive for the `*(l_extendedprice,+(1,l_tax))` expression generated from the snode tree from Figure 5.2. Code redundancies are dealt with by the C compiler optimizer. Indentation and comments are added for clarity.

```
uidx r0 (uidx n, sint * __r res, void **params, uidx * __r sel, void *e) {
  uidx i;
  sint r4; ulng e5 = 0; /* temporary result variable and error flag */
  sint *const v14 = (sint *) params[0]; /* params[0] is a vector of l_extendedprice */
  schr r10; ulng e11 = 0; /* temporary result variable and error flag */
  r10 = 100; /* the constant 100 */
  schr r12; ulng e13 = 0; /* temporary result variable and error flag */
  schr *const v15 = (schr *) params[1]; /* params[1] is a vector of l_tax */
  schr r8; ulng e9 = 0; /* temporary result variable and error flag */
  sint r6; ulng e7 = 0; /* temporary result variable and error flag */
  sint r2; ulng e3 = 0; /* temporary result variable and error flag */
  if (sel) { /* variant with a selection vector */
      for (i = 0; i < n; i++) {
          r4 = v14[sel[i]]; r12 = v15[sel[i]]; /* read from input vectors */
          r8 = ((r10) + (r12)); /* 100+l_tax */
          r6 = ((sint) (r8)); /* cast to 4-byte integer */
          r2 = ((r4) * (r6)); /* multiply by l_extendedprice */
          res[sel[i]] = r2; /* save to result vector */
      }
  } else { /* variant without a selection vector */
      for (i = 0; i < n; i++) { /* same operations as above */
          r4 = v14[i]; r12 = v15[i]; /* read from input vectors */
          r8 = ((r10) + (r12)); /* 100+l_tax */
          r6 = ((sint) (r8)); /* cast to 4-byte integer */
          r2 = ((r4) * (r6)); /* multiply by l_extendedprice */
          res[i] = r2; /* save to result vector */
      }
  }
  /* Error checking (even though the operations don't produce errors */
  if (e9) return x100_err(e, VWLOG_FACILITY, X100_ERR_GENERIC);
  if (e7) return x100_err(e, VWLOG_FACILITY, X100_ERR_GENERIC);
  if (e3) return x100_err(e, VWLOG_FACILITY, X100_ERR_GENERIC);
  return n; /* return number of processed tuples */
}
```

Algorithm 16 shows a JIT primitive for the example expression `*(l_extendedprice, +(1,l_tax))`, generated from the `snode` tree from Figure 5.2. It can be seen that there is some redundant code generated. For example, the error variables are always checked, even if the operation did not use them. The same however happens in normal primitives, and there's no possibility to detect it when generating the source template. Another example is that there are a lot of redundant local variables created. A value from an array is first read into a local variable before being processed. Even such simple operations like casts are very explicit, assigned into a new local variable. This is however because they also had to be explicit in the query execution tree, where whole vectors were cast from one type into another.

The C compiler is able to detect and reduce all these redundancies in simple optimization passes. Undoubtedly, this introduces more compilation overhead, both because of the bigger volume of C code that needs to be parsed and by more work for the C compiler optimizer. However, we have already shown in Chapter 4 that compilation overhead is minor, and such implementation enables us to simplify code generation significantly.

5.4. Primitives storage and maintenance

After being compiled, the JIT primitives are stored in a hash table for future reuse. Later, if a `snode` tree is built for the same combination of operations, the old primitive could be looked up and reused instead of generating source and compiling again.

To be able to lookup a JIT primitive in a hash table, it needs to have a string name that unambiguously describes it. We can create such a name by walking through the `snode` tree and outputting the names of particular nodes pre-order (left deep). The name of individual nodes come from the name of the source templates for operation nodes, types of input parameters for input nodes, constant values and types for constants etc. It leads to unambiguous naming scheme that may not be beautiful, but serves the purpose of naming the primitives, so that they can compared and looked up in a hash table.

The JIT primitive from Algorithm 16 has the name:
`jit_map__*_sint_sint(sint_col0,_sint_schr(_+_schr_schr(schr_100,schr_col1)))`

There is a system table that can be queried to see all JIT primitives that are present in the system. It can be done using:

```
SysScan('jit_primitives', ['name', 'return_type', 'ptr']);
```

There is also a system call in VectorWise that flushes all existing buffered JIT primitives from the system. This should unload all dynamically loaded libraries and free object code memory. It can be done using:

```
Sys(clear_jit);
```

5.5. Summary

In this chapter we have shown how an implementation of JIT compilation infrastructure and code generation for compiled execution of parts of a query can be done in a way that does not require big modifications to an existing system. With relatively little code we are able to generate code covering a large class of operators. Still, in the scope of this thesis we were not able to implement all our ideas, and further extensions remain future work.

Chapter 6

JIT primitives evaluation

In this chapter we present some benchmarks of how the implemented JIT compilation affects the query execution performance of a working VectorWise system and compare it to the theoretical benefits and hard-coded benchmarks discussed in Chapter 3.

We evaluated the implementation of JIT compiled primitives in the TPC-H benchmark. We used a computer with 2.67 Ghz Nehalem processor and 12 GB of RAM. We used TPC-H scale factor 10, which corresponds to a 10 GB database. We run the queries without parallelism and we measure "hot" runs, with the database loaded into the buffer pool so we read from RAM instead of disk, so that the queries are more CPU intensive and less I/O intensive.

Due to limitations of the Rewriter based implementation of the JIT compilation logic, the change in system performance on the TPC-H benchmark as a whole is limited. We are able to detect and compile together:

- Arithmetic calculations that were evaluated using map primitives. They are mostly seen in Project operator, but are also possible e.g. in complicated selection conditions.

- Selection conditions generating a selection vector for the Select operator. It can generate code for both eager and lazy evaluation of the conditions. Multiple Select operators can be combined into one, with just one huge compiled condition evaluation function.

This has an impact on a few of the TPC-H queries: Query 1, Query 6, Query 12 and Query 19. An overview of JIT compilation effect on the whole queries is shown in Table 6.1. In the following sections we will discuss each of them in detail.

TPC-H Scale Factor 10		
Query	Without JIT	With JIT
1	1.93 s	1.79 s (-7.2%)
6	0.15 s	0.21 s (+40%)
12	0.50 s	0.43 s (-14%)
19	1.36 s	1.05 s (-22.8%)

Table 6.1: Overview of TPC-H Queries 1, 6, 12 and 19 at scale factor 10 with and without JIT compilation.

TPC-H Scale Factor 10, Query 1		
Projection	Tot. cycles (vect.)	Tot. cycles (JIT)
Total:	1,167,793,680	**781,728,432 (-33.1%)**

Table 6.2: Compiled projection in TPC-H Query 1.

TPC-H Scale Factor 10, Query 6			
Selection	Sel.	Tot. cycles (vect.)	Tot. cycles (JIT)
`l_shipdate < date('1995-01-01')`	87.43%	32,880,796	
`l_quantity < 24`	45.97%	58,692,544	
`l_discount >= 0.05`	54.53%	28,413,788	
`l_shipdate >= date('1994-01-01')`	85.85%	12,343,920	
`l_discount < 0.07`	49.94%	15,621,912	
Total:	9.40%	147,952,960	**296,671,052 (+100.5%)**

Table 6.3: Compiled selections in TPC-H Query 6.

6.1. TPC-H Query 1

This query was already presented in Algorithm 13 on page 68. In this query we are able to compile together the computation of arithmetic expressions in a projection. These expressions have already been the inspiration for the Project benchmark in Section 3.2 of Chapter 3.

Since the expression `l_extendedprice * (1 - l_discount)` is reused by two aggregate computations, we separately compile this expression and later the expression `res * (1 + l_tax)`. It could be more effective if primitives returning more than one return vector would be implemented, because then it could have been compiled together.

Overall, it speeds up the Project operator in Query 1 by 33%, as is shown in Table 6.2. However, the most time consuming part of this query is the aggregation, so it does not translate to such a big speedup of the whole query, which is just over 7%.

6.2. TPC-H Query 6

This query is presented in Algorithm 17. It demonstrates well the results of the Conjunctive Selection benchmark from Section 3.3. The query consists of a single aggregate over the `lineitem` table filtered with four simple selection conditions, each being implemented as an integer comparison. We focus on those selections, which together take 57.75% of query execution time, another 30% being Scan, and only a few percent Aggregation.

In vanilla VectorWise, the selections are implemented by four separate Select operators stacked on top of each other. The JIT compilation rules combine them all into one operator evaluating all the conditions by a single primitive. The generated JIT function evaluates the condition lazily with nested "if" branches, as in the benchmarks from Section 3.3. The result is that it actually makes the selection twice slower due to branch mispredictions! This is shown in Table 6.3. All of the filter conditions have medium selectivities and the conditions themselves are computationally cheap, so this result confirms the previous hard-coded benchmarks. Since the selection corresponds to a substantial part of query execution time, this makes a 40% slowdown of the whole query.

This is one of the faster running queries, so that does not have a big impact on the total TPC-H benchmark performance result. However, it signals the need of implementing a

TPC-H Scale Factor 10, Query 12			
Selection	Sel.	Tot. cycles (vect.)	Tot. cycles (JIT)
`l_shipmode in ('MAIL', 'SHIP')`	28.58%	**500,450,608**	
`!=_str_col_str_val()`		180,805,384	
`!=_str_col_str_val()`		180,971,184	
`||_lazy_adaptive_uidx_col_uidx_col()`		132,944,656	
`l_receiptdate < date '1995-01-01'`	85.22%	18,080,888	
`l_receiptdate >= date '1994-01-01'`	82.70%	13,991,616	
`l_shipdate < l_commitdate`	48.72%	18,023,004	
`l_commitdate >= l_receiptdate`	24.54%	9,831,664	
Total:	2.41%	560,377,780	**361,024,540 (-35.6%)**

Table 6.4: Compiled selections in TPC-H Query 12.

TPC-H Scale Factor 10, Query 19			
Selection	Sel.	Tot. cycles (vect.)	Tot. cycles (JIT)
`l_shipmode in ...`	14.28%	1,653,803,608	
`l_shipinstruct == ...`	25.00%	197,060,672	
`l_quantity between 1 and 30`	59.96%	252,331,068	
Total:	2.41%	2,103,195,348	**1,308,809,160 (-37.8%)**
`p_brand in ...`	11.99%	104,048,228	
`p_container in ...`	30.02%	53,450,032	
`p_size < 15`	29.98%	3,657,288	
`p_size >= 1`	100.00%	305,680	
Total:	2.41%	161,461,228	**232,642,600 (+43.4%)**
Selection above join:	8.17%	110,356,512	**12,336,860 (-88.8%)**
Total:		2,375,013,088	**1,553,788,620 (-34.6%)**

Table 6.5: Compiled selections in TPC-H Query 19.

smarter criterion for evaluating the feasibility of JIT compilation and not compiling naively as much as possible.

6.3. TPC-H Query 12

This query is presented in Algorithm 17. Here we also have a stack of selection conditions, however the first Select operator executing `l_shipmode in ('MAIL', 'SHIP')` is dominating the whole execution time. Because of that, in Table 6.4 we further split the execution profile of this operator into individual primitives: the two string comparison primitives, and the primitive that controls lazy execution and negates the selection vector of a disjunction (as discussed in vectorized lazy disjunctions implementation in Section 3.4).

The first Select operator is a disjunction of two string comparisons: `l_shipmode = 'MAIL' or l_shipmode = 'SHIP'`. It processes 13 million tuples from the `lineitem` table. There are two reasons why such a disjunction can work better compiled together as a JIT primitive:

Algorithm 17 Query 6 and 12 of the TPC-H benchmark.

```
# Query 6
select
        sum(l_extendedprice * l_discount) as revenue
from    lineitem
where   l_shipdate >= date '1994-01-01'
        and l_shipdate < date '1994-01-01' + interval '1' year
        and l_discount between 0.06 - 0.01 and 0.06 + 0.01
        and l_quantity < 24;

# Query 12
select
        l_shipmode,
        sum(case when o_orderpriority = '1-URGENT'
                        or o_orderpriority = '2-HIGH'
                        then 1
                 else 0
        end) as high_line_count,
        sum(case when o_orderpriority <> '1-URGENT'
                        and o_orderpriority <> '2-HIGH'
                        then 1
                else 0
        end) as low_line_count
from    orders, lineitem
where   o_orderkey = l_orderkey and l_shipmode in ('MAIL', 'SHIP')
        and l_commitdate < l_receiptdate and l_shipdate < l_commitdate
        and l_receiptdate >= date '1994-01-01'
        and l_receiptdate < date '1994-01-01' + interval '1' year
group by l_shipmode
order by l_shipmode;
```

- The evaluation of a string comparison condition is expensive in itself. It involves looping over strings and breaking out of the loop after finding a mismatching character. This causes branch mispredictions and suboptimal processor utilization even in the pure vectorized implementation, so compiling them together should not bring further damage.

- Implementation of lazy disjunctions requires to execute them as negated conjunctions. Negating the condition involves creating an inverted selection vector, which is a branch heavy and expensive operation, as was discussed in Section 3.4. In this particular example we can see from the profile in Table 6.4 that the part of the operator controlling lazy execution and inverting the selection vectors takes 26% of the operator's execution time. Compiled implementation can implement disjunction directly and save all this overhead.

A single JIT primitive that compiles together all the selection conditions of this query is as fast as the string comparison primitives alone were in the vanilla vectorized execution. Overall, it speeds the Select operators by over 35% and the whole query by 14%.

6.4. TPC-H Query 19

This query was already presented in Algorithm 13 on page 68. It is another query with a lot of disjunctive selection conditions on strings: `p_brand`, `p_container` and `l_shipmode` are each compared to a number of string constants.

The selection conditions relate to attributes from two joined tables: `lineitem` and `part`. In query execution they have to be separated into three selections: one pre-selection relating

to one table for each side of the join and a post-selection after the join with conditions on attributes from both tables. In Table 6.5 we show each of these operators separately.

The preselection on the `lineitem` table part dominates the overall execution time. It processes almost 6 million tuples from that table. The result here is very similar to that from query 12. The JIT combined primitive is able to speed up disjunctive string comparison selection on `l_shipmode`, which speeds the whole operator up by 38%.

However, even though the situation on the other side of the join seems similar, we get a slowdown instead of speedup with JIT compilation there. From the `part` table 2 million tuples are processed. There are two disjunctive string selections on `p_brand` and `p_container` that dominate execution time on this side of the join. In this case however, compiling them together into one JIT primitive turned out to be 40% slower than executing it in vanilla VectorWise.

The Select operator above the join contains the whole huge condition on both tables. In vanilla VectorWise it uses 45 separate primitives to evaluate it. JIT compilation is able to combine them together into one gigantic JIT primitive. We can see that it speeds up execution of this operator by 88%! It does not have impact on the whole query however, as only several thousand tuples are processed above the join.

The three Select operators are together faster by almost 35%, even though the preselection on `part` table side is significantly slower. The whole query is executing almost 23% faster with JIT compilation.

6.5. Summary

The evaluation confirmed our previous experiments and suspicions, that naive compilation of everything that we can generate code for is not always the best answer. Cost models will have to be carefully developed to make the VectorWise system automatically able to decide what strategy is the best. The impact of JIT compilation on full TPC-H was limited, since the implementation is still basic. More possibilities are promised by the experiments from Chapter 3. For now, some of the results we obtained are already promising.

Chapter 7

Conclusions and future work

7.1. Contributions

In this master thesis we conducted an analysis of potential benefits of combining the techniques of vectorization and compilation in a database query execution engine. We identified situations where such a combination shows substantial improvements. Existing database engines, either scientific or commercial, concentrated on only one of these two methods. Our main message is that one does not need to choose between compilation and vectorization. The study in Chapter 3 described concrete examples and use cases that directly show that.

We applied our findings to the VectorWise DBMS by adding a JIT query compilation infrastructure to the system. This infrastructure forms a solid base for JIT compilation experiments and evaluation. In the scope of the thesis we were able to implement detection and code generation for simple operations such as arithmetic calculations in projections or condition evaluation in selections.

7.2. Answers to research questions

Our research questions were:

- Since a vectorized systems already removed most of this overhead, would it still be beneficial to introduce JIT query compilation to it?
- Can benefits of vectorization still make an improvement to a tuple-at-a-time compiled system?

Our study answered both of these questions with a "yes" answer. We observed speedups of 7% to 23% on TPC-H queries 1, 12 and 19 by only compiling limited small fragments of them. When counting only the fragments that we actually compiled, the speedups were as high as 30% to 35%. While the difference is not as stunning as when introducing vectorization or compilation to a plain tuple-at-a-time system, this is achieved with a very simple implementation. Chapter 3 suggests more opportunities in e.g. hash operations. We anticipate that speedups of as much as 50% can still be gained in some situations.

The thesis has also shown some very interesting results concerning differences between compilers. These differences are easy to overlook when examining the performance of entire queries, as study in Chapter 4 has shown that the overall TPC-H benchmark result of the VectorWise DBMS compiled by clang, gcc or icc differs by no more than 5%. However, the focus of JIT compilation is on small fragments of operators. When examining them in detail,

as we have done in Chapter 3, we can see that the compilation result of different compilers can vary significantly. While the choice of a compiler for the VectorWise system as a whole does not matter that much, some additional benefits can be gained by choosing the best compiler for the given situation with fine granularity.

7.3. Future work

Work on JIT compilation in the VectorWise system will continue. An important next step is to expand JIT opportunities detection capabilities. As mentioned in Section 5.2.4, this will require an alternative implementation based in the Builder phase of query processing instead of Rewriter. Adapting this will be the first major upcoming goal. Foreseeing this, the current implementation separates this phase from code generation, so only the `snode` tree construction will have to be adjusted.

Having a new JIT compilation detection implementation, we can work on providing code generation for other ideas from Chapter 3.

- Compiled hash tables. This will affect not only HashJoin as described in Section 3.6, but also other operators that use hashing, such as Aggregation. Introducing compiled hash tables would require some intrusive changes, as right now the hash table logic is split between the hash table implementation itself and the operators using them.

- For Aggregation we can combine computing multiple aggregates together. This way, we would ease the data dependency when the same group is accessed in subsequent iterations. That would require implementing Expressions that can return multiple result vectors.

- Operators like sorting could benefit from compilation. Right now there is no block oriented way of doing sorting, and query execution has to fall back to interpreted tuple-at-a-time processing.

Another field for further work is establishing a good heuristic or a cost model to make decision about when compilation is beneficial. It was shown in Chapter 3 that compilation is not always the best answer, and it was seen in live-system examples in Chapter 6 that the naive greedy algorithm of always compiling the largest component can bring damages instead of benefits. A cost model should take into account not only the expected performance of compiled function versus that of pure vectorized primitives, but also other factors like compilation overhead (significant for fast running queries) or reuse potential (maybe better to compile a few smaller fragments that could repeat in future queries than one monolithic function).

We could also think about JIT primitives buffering policy. Right now we are only able to flush all the JIT object code clear. We could think of counting how much the primitives are being reused and evict them selectively. This may however prove to be a hard task, because of primitives coming from dynamically loaded libraries, multiple primitives in one library from one compilation, and no reliable way to estimate the memory usage of individual primitives.

Acknowledgments

This master thesis was made possible thanks to Marcin Żukowski and the VectorWise company where I was offered an internship.

I would like to thank Marcin Żukowski and Peter Boncz, who were my supervisors and supported me daily. With my stubborn nature, listening to their remarks, comments and ideas were one of the rare moments in my life when I really felt I truly and directly learn a lot. They pushed my ambitions and encouraged me to believe in my skills not only in programming, but also in writing and giving talks. Thanks to them, this thesis resulted in a conference publication and I could enjoy a week long trip to Athens.

I would like to thank my second reader Henri Bal for contributing to the final shape of this thesis.

Michał S. offered me a lot of help when I was starting the project and getting to know the VectorWise system. His patience and answers to all my questions were really of great help. Thanks to him for that, and for some great time after work.

I have been working in a very friendly environment and I have learned a lot here. I would like to thank Gosia, Giel, Hui, Sandor, Willem and Irish for contributing to the great time I spent in the VectorWise office.

Michał Ś., my fellow co-intern shared the office with me the whole time and was a sounding board for all my everyday problems and ideas. Thank you for coping with that.

I would like to thank Kamil, who did his master thesis at VectorWise last year and continued working here, and from whom I learned about this company.

It is Ala without whom I would not even be in The Netherlands. She motivated me to apply for the Short Track Masters programme and come to Vrije Universiteit Amsterdam for my final year of studies. It was her idea, and it turned out well for both of us. We spent this year almost 24 hours a day together. Living in one room, attending the same university courses, both working on our theses at VectorWise. I thank her for being with me all this time.

Last but not least, I would like to thank my family for their everlasting support.

Bibliography

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[Beg08] Nate Begeman. Building an Efficient JIT. In *LLVM Developers' Meeting*, 2008.

[Bon02] P. A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. Ph.d. thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, May 2002.

[BZN05] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. CIDR*, Asilomar, CA, USA, 2005.

[CK85] A. Copeland and S. Khoshafian. A Decomposition Storage Model. In *Proc. SIGMOD*, Austin, TX, USA, 1985.

[cla] *Clang*. `http://clang.llvm.org`.

[D. 81] D. Chamberlin et al. A history and evaluation of System R. *Commun. ACM*, 24(10):632–646, 1981.

[Fog11] Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs. An optimization guide for assembly programmers and compiler makers*, 2011.

[GBC98] Goetz Graefe, Ross Bunker, and Shaun Cooper. Hash Joins and Hash Teams in Microsoft SQL Server. In *VLDB*, pages 86–97, 1998.

[Gra94] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE TKDE*, 6(1):120–135, 1994.

[Gre99] Rick Greer. Daytona and the fourth-generation language Cymbal. In *SIGMOD*, 1999.

[Gro10] Tobias Grosser. Polly - Polyhedral optimizations in LLVM. In *LLVM Developers' Meeting*, 2010.

[GZA+11] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly - Polyhedral optimization in LLVM. In *IMPACT*, 2011.

[KN10] Alfons Kemper and Thomas Neumann. HyPer: Hybrid OLTP and OLAP High Performance Database System. Technical report, Technical Univ. Munich, TUM-I1010, May 2010.

[Kri10] Konstantinos Krikellas. The case for holistic query evaluation. 2010.

[KVC10] Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. Generating code for holistic query evaluation. In *ICDE*, pages 613–624, 2010.

[LLV] LLVM. *The LLVM Compiler Infrastructure*. `http://llvm.org`.

[Nau10] Axel Naumann. Creating cling, an interactive interpreter interface for clang. In *LLVM Developers' Meeting*, 2010.

[Par10] ParAccel Inc. Whitepaper. *The ParAcel Analytical Database: A Technical Overview*, Feb 2010. `http://www.paraccel.com`.

[PMAJ01] S. Padmanabhan, T. Malkemus, R. Agarwal, and A. Jhingran. Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *Proc. ICDE*, Heidelberg, Germany, 2001.

[Ros02] Kenneth A. Ross. Conjunctive selection conditions in main memory. In *Proc. PODS*, Washington, DC, USA, 2002.

[RPML06] Jun Rao, Hamid Pirahesh, C. Mohan, and Guy M. Lohman. Compiled Query Execution Engine using JVM. In *Proc. ICDE*, Atlanta, GA, USA, 2006.

[sse] *Intel SSE4 Programming Reference*. `http://software.intel.com/file/18187/`.

[SZB11] Juliusz Sompolski, Marcin Zukowski, and Peter A. Boncz. Vectorization vs. Compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware, DaMoN 2011*, pages 33–40, 2011.

[TJYD09] D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting performance data with PAPI-C. *Tools for High Performance Computing*, pages pp. 157–173, 2009.

[Tom] Tom. *Tom*. `http://tom.loria.fr/`.

[Tra02] Transaction Processing Performance Council. *TPC Benchmark H version 2.1.0*, 2002.

[Vec] VectorWise. *VectorWise*. `http://www.vectorwise.com`.

[ZBNH05] M. Zukowski, P. A. Boncz, N. J. Nes, and S. Heman. Monetdb/X100 - A DBMS In The CPU Cache. *IEEE Data Engineering Bulletin*, 28(2):17 – 22, June 2005.

[ZNB08] Marcin Zukowski, Niels Nes, and Peter Boncz. DSM vs. NSM: CPU Performance Tradeoffs in Block-Oriented Query Processing. 2008.

[Zuk09] Marcin Zukowski. *Balancing Vectorized Query Execution with Bandwidth-Optimized Storage*. Ph.D. Thesis, Universiteit van Amsterdam, Sep 2009.