Coldbase - A Column-Oriented In-Memory Database

Johan Jonsson

February 10, 2009
Master's Thesis in Computing Science, 30 ECTS-credits
Supervisor at CS-UmU: Michael Minock
Examiner: Per Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

General row-oriented SQL databases provide a wide range of functionality which in practice causes an unacceptable space and performance penalty to some applications. Within the financial sector the typical workload is often a large timestamped ordered input data stream and a small set of simple, but frequent aggregations over a few columns. By creating a specialized implementation that focuses on this type of workload several improvements to performance can be made.

This paper introduces a new multi-threaded Java based implementation of a column oriented database. It takes advantage of the expected workload using multiple threads and tailoring the concurrency control to a suitable granularity. It also provides a extendable interface and takes advantage of the integration capabilities that Java is known for. The potential performance and memory overhead sometimes associated with Java applications is also addressed by focusing on using primitive types.

Benchmark results demonstrate the performance potential of Coldbase. The evaluation shows that it outperforms a commonly used row-oriented database for the typical workload and discusses future improvements to how additional performance can be gained.

Contents

1 Introduction **1**
- 1.1 Report outline . . . 1
- 1.2 Problem Description . . . 1
  - 1.2.1 Nomura . . . 2
  - 1.2.2 Goals . . . 2
  - 1.2.3 Restrictions . . . 3
  - 1.2.4 Methods . . . 3
  - 1.2.5 Related Work . . . 3

2 Column-oriented databases **5**
- 2.1 Row orientation vs. column orientation . . . 5
  - 2.1.1 Conditions and aggregations vs. random access . . . 6
  - 2.1.2 Compression . . . 6
  - 2.1.3 Inserts, updates and deletes . . . 6
- 2.2 Vectorization . . . 7
- 2.3 Insert, update and delete techniques . . . 8
  - 2.3.1 Inserts, updates and deletes in C-store . . . 9
- 2.4 Compression . . . 9
  - 2.4.1 Operating directly on compressed data . . . 10
  - 2.4.2 Cache-aware decompression . . . 11
  - 2.4.3 Selecting compression strategy . . . 11
- 2.5 Vertical partitioning vs. column-orientation . . . 12

3 Java performance **15**
- 3.1 Memory overhead . . . 15
  - 3.1.1 Java strings . . . 16
- 3.2 Loops over primitive types . . . 17
- 3.3 Conclusions . . . 18

4 Implementation **19**
- 4.1 Data types and representations . . . 19

4.1.1 Vectors 19
4.1.2 Atoms 20
4.1.3 Numeric types 21
4.1.4 Type conversions and coercions 22
4.1.5 Date and time 23
4.1.6 Characters and Strings 23
4.1.7 Generic types 24
4.1.8 Indexing and finding 24
4.1.9 Bit vectors 25
4.1.10 Null value representations 25
4.2 Map 26
4.3 Tables 28
4.3.1 Left outer join 28
4.4 Queries 29
4.4.1 Conditions 30
4.4.2 Aggregates 31
4.4.3 Group-by 31
4.5 Multi threading 33
4.5.1 Concurrency in Coldbase 33
4.5.2 Locking facility 34
4.5.3 Appends to keyed tables 34
4.5.4 Concurrency in tables and queries 34

5 Results **35**
5.1 Introduction to Coldbase usage 35
5.1.1 Creating tables and inserting data 35
5.1.2 Making queries over the table 35
5.1.3 Threading and synchronization 36
5.1.4 Grouping 36
5.2 Interface and API evaluation 37
5.3 Code maintainability 37
5.4 Query performance 38
5.4.1 Finding the bottlenecks 38
5.4.2 Possible solutions 39
5.5 Multi threading 40
5.5.1 Possible improvements 42

6 Conclusions **43**
6.1 Limitations 43
6.2 Future work 44

7 Acknowledgements **45**

References **47**

A User's Guide **49**
A.1 Vectors . . . 49
A.1.1 Common interface . . . 49
A.1.2 Numeric vectors . . . 50
A.1.3 Time vectors . . . 51
A.1.4 Date vectors . . . 51
A.1.5 String vector . . . 51
A.1.6 Bit vector . . . 52
A.1.7 Char vector . . . 52
A.1.8 Vector of vectors and generic vectors . . . 52
A.1.9 Additions to integer vectors . . . 52
A.1.10 Examples . . . 52
A.2 Tables . . . 53
A.2.1 Keyed tables . . . 54
A.2.2 Examples . . . 54
A.3 Manual synchronization . . . 54
A.3.1 Examples . . . 55
A.4 Query interface . . . 56
A.4.1 Operations . . . 56
A.4.2 Conditions . . . 56
A.4.3 Grouping . . . 57
A.4.4 Examples . . . 57
A.5 Utilities . . . 58
A.5.1 Printing tables . . . 58
A.5.2 Timing queries . . . 58

B Benchmarks and results **59**
B.1 Quote data . . . 59
B.2 Primitive type loops in Java and C++ . . . 59
B.3 Hash map benchmark . . . 59
B.3.1 Key map . . . 60
B.3.2 Index map . . . 61
B.4 Database benchmarks . . . 63
B.4.1 Query with condition . . . 63
B.4.2 Simple aggregate . . . 64
B.4.3 Grouped aggregates . . . 64
B.4.4 Left outer join . . . 64
B.4.5 Results . . . 65
B.5 Multi-threading benchmarks . . . 65
B.5.1 Primitive vector operations . . . 65

B.5.2 Aggregate query with groups 66
B.5.3 Another aggregate query with groups 67
B.5.4 Left outer join 67
B.5.5 Results 67

C Source Code 69
C.1 Bit vector `where` 69
C.2 IntVector *less than* method, `lt` 70

List of Figures

2.1 Binary vector operator. 7
2.2 Null suppression example. 10
2.3 Decompression to cache vs. to RAM. 11

3.1 Benchmark results for operations on primitive type arrays. 18

4.1 Inheritance graph for vector types. 20
4.2 Example of `geq` between vector and atom. 21
4.3 Map insert, step 1. 26
4.4 Inheritance graph for the query system 30

5.1 Query tree where data is not shared. 40
5.2 Query tree where data is shared. 40
5.3 Primitive operations benchmark. 41

B.1 Benchmark `KeyMap` vs. `HashMap`. 61
B.2 Overlapping benchmark of `KepMap` vs. `HashMap`. 62
B.3 Benchmark `IndexMap` vs. `HashMap`. 62
B.4 Overlapping benchmark of `IndexMap` vs. `Hashmap`. 63
B.5 Query benchmark. 65
B.6 Query benchmark excluding PostgreSQL. 66
B.7 Multi-threading benchmark, Sun UltraSparc. 68
B.8 Multi-threading benchmark, Intel Core 2. 68

List of Tables

2.1 Database table with names. 5
2.2 Result from vertical partitioning of Table 2.1. 12

3.1 Memory consumption of Java `String`. 17

4.1 Coercion order of numeric types. 22
4.2 Null values for primitive types. 25
4.3 Example of left outer join, *source* `loj` *lookup*. 29
4.4 Example of atom expansion for aggregates. 31
4.5 Example of group by 32
4.6 Illustration of map when performing group-by. 32
4.7 Lock level compatibility. 33

5.1 Table with timestamped transactions. 36
5.2 Primitive operations benchmark. 42

B.1 Excerpt from quote data. 60
B.2 Benchmark results for operations on primitive type arrays. 60
B.3 Benchmark times in milliseconds of sample queries. 66

List of Listings

2.1 Vectorized equality operation. 8
3.1 String caching example. 17
4.1 Example vector operation. 22
4.2 Example query. 30
4.3 Example condition query 30
4.4 Example group-by query 32
5.1 Creating a keyed table 35
5.4 Element-wise division methods. 41
A.1 Example of automatic coercion of numeric types. 52
A.2 More extensive example for manipulating dates. 53
A.3 Example of how to create a simple keyed table. 54
A.4 Example of how to create a simple table from preexisting data. 54
A.5 Example of how to externally synchronize vectors. 55
A.6 Example of locking and unlocking multiple locks. 55
A.7 Example query with a simple condition. 57
A.8 Example query with groupings. 57
A.9 Example query rounding times. 57
A.10 Example query with nested operations. 57
A.11 Example of printing tables. 58
A.12 Example of query timing. 58
B.1 SQL for the condition query benchmark. 63
B.2 Java code for the condition query benchmark. 64
B.3 Java code for the aggregate query benchmark. 64
B.4 Pseudo SQL for benchmark of aggregates over grouped data. 64
B.5 SQL for left outer join benchmark. 65
C.1 Implementation of `where`. 69
C.2 Excerpt from implementation of `lt` in `IntVector`. 70

Chapter 1

Introduction

This thesis introduces an in-memory column-oriented database constructed with a given workload profile in mind. The workload typically includes a large input stream of data ordered by timestamp and simple but frequent analytical queries. Optimizing towards a specific workload opens up design possibilities not available to a general purpose design. General functionality is included as well, but potentially with a performance penalty in some cases.

Coldbase is written entirely in Java. The main benefit is the ease of integration with other Java based systems and platform independence while potential disadvantages might include memory overhead and performance penalties. The design aims at minimizing the memory overhead and maximizing the performance.

1.1 Report outline

In the following sections of this chapter the problem is described, the goals of the thesis are formulated and the restrictions on the implementation are listed. In Chapter 2 a general study of column-oriented databases is presented. It covers a few common strategies and a few methods that are unique to some of the current implementations.

In Chapter 3 a few tests are made to establish if Java could potentially provide sufficient performance for this type of application. A small benchmark comparing a C++ program to a Java program is presented and discussed. The implementation is presented in Chapter 4. The core data structures and a few of the most interesting algorithms are described. The results and evaluation of the implementation are discussed in Chapter 5. It also covers a small introductory section to how the database interface can be used. The results include a brief evaluation of the interfaces, a discussion of the benchmark results and a discussion of where the bottlenecks lie and possible solutions. Finally in Chapter 6 the final conclusions of the thesis, the limitations, and future work is discussed.

1.2 Problem Description

Column-oriented databases are becoming increasingly popular in areas where fast aggregated queries are required large amounts of data. In the financial sector, typical data streams contain timestamped data which are inserted at the end of the ordered primary

key column during the day. The amount of data is substantial, often many thousands of tuples per minute.

To make fast and accurate estimates on how the market is behaving it is critical to be able to execute fast analysis on the trading data. Such analysis often results in frequent large bulk selects, or simple aggregations over very few of the columns with simple logic. There is generally an absence of complex joins and most queries are written by programmers acquainted with the implications of executing a query. The queries typically acts as filters, selecting sparsely from a large space and performing some simple aggregations. Using the knowledge about typical queries the performance can be optimized towards specific workloads. Since queries are typically written by programmers, automatic query optimization is not critical.

Typical SQL-databases provide much wider functionality than is required, which causes an unacceptable space and performance penalty. This is the typical case where the column oriented design is beneficial.

This project aims at designing and implementing a Java based in-memory column-oriented database and evaluating its performance. It will be optimized for appends, i.e. adding data at the end of the structure, and it will provide multi-threading support with coarse grained synchronization and multi-level locking. The greatest disadvantage with using Java is the potential performance and memory penalty. The benefits include platform independence, a rich API, but most importantly *ease of integration* with existing and new applications.

1.2.1 Nomura

This thesis project is based on a proposal from the Nomura company (former Lehman Brothers). Typical database schemas, data, and queries are known beforehand and was provided by Nomura. The supplied data and queries should reflect some of their workloads and can be used to simulate real usage. The design will primarily be focused towards getting good performance for this data and workload. General functionality is provided as well, but potentially with a more significant overhead.

1.2.2 Goals

The primary goal is to get performance that can be compared with similar products while keeping the interface clean, extendable and easy to use. More specifically, the most important cases for which performance is to be considered are:

– Appending data to tables in parallel with reading from the same table.

– Performing multiple reads from the same table simultaneously.

– Executing fast aggregations over columns with simple logic.

– Executing fast aggregations over grouped results.

Other goals include evaluation of using Java for this type of application. Both the performance aspects and the design implications that a Java implementation can impose should be evaluated.

1.2.3 Restrictions

To limit the scope appropriately some restrictions are necessary:

- The prototype need only contain a sufficient set of functions to be able to evaluate the performance. It must however be easy to add functionality at a later time.
- The prototype is in-memory only.
- No transactions are implemented.
- Most of the optimizations and design choices are made with focus on the provided data and typical workload.

1.2.4 Methods

To reduce the memory overhead typically imposed by Java, data must be stored in its primitive form as opposed to being wrapped by objects. Primitive types in Java do not add any additional memory overhead and looping over arrays of primitive types is very efficient. The problem is that Java generics does not allow parameterizing primitive types without wrapping the data (autoboxing) into objects, and there is no built-in preprocessing functionality.

Vector based operations can benefit from execution facilities of modern CPU's and while operating on primitive arrays the amount of method calls can also be relatively low. This will be used throughout the design to get good performance. Also as of Java HotSpot[1] VM 1.3, loop unrolling, instruction scheduling and range check elimination are built in features which should provide performance closer to lower-level languages[10].

For the evaluation part, the performance will be compared to a well known column oriented product, as well as a row-oriented database.

1.2.5 Related Work

Related products today include KDB[2], MonetDB[16], Sybase IQ[13], C-Store[15] and others. None of these products are developed in Java. More on how these products implement different features can be found in Chapter 2.

[1] Sun's Java HotSpot implements the Java Virtual Machine Specification and is supplied with the Java SE platform. See `http://java.sun.com/javase/technologies/hotspot/index.jsp` for more information

Chapter 2

Column-oriented databases

This chapter covers a literature study focusing on designs of existing column-oriented databases and other related work.

2.1 Row orientation vs. column orientation

The database orientation is a physical property of the database. Conceptually a database table is a two dimensional structure. At the physical level however, it needs to be mapped onto the one dimensional structure of the computer memory address space before being stored.

The mapping from tables onto a one dimensional structure can be done in two obvious ways: row-wise or column-wise. Consider Table 2.1. Storing this table row-wise will group the information in each *record*, in this case each person, together. The information will be consecutively stored, person by person. A column oriented structure would instead store each *attribute* together, first all the ids, then all the first names and finally all the last names.

Table 2.1: Database table with names.

id	fname	lname
524	John	Smith
525	Jane	Grant
530	Bill	Johnson

Both mappings has its advantages and disadvantages. The design choice is based on what types of queries are expected. If typical queries tend to access the data on a record basis, it is good to have records consecutively stored and a row-oriented approach would be preferred. On the other hand, if some kind of analysis over one or a few attributes is made, such as for example finding the most common first name, it is good to have the columns consecutively stored and a column-oriented design is better. Row-oriented databases are instead equipped to provide good performance for these type of queries in the form of indexes, using for example B-trees.

The following sections will describe the basic implications of choosing a column-oriented design over a row-oriented design, including both advantages and disadvantages.

2.1.1 Conditions and aggregations vs. random access

The condition in a query, the `WHERE`-clause in SQL, is a boolean expression which will filter records from a table. In many cases this expression only addresses one or a few of the columns of the table. For column-oriented databases this means that only the addressed columns will be read and scanned. For row-oriented databases however, the whole table must be scanned, unless an index on the column is available. Keeping indexes, on the other hand, introduces more bookkeeping as well as storage overhead, and if the workload is not predictable it is hard to decide what the indexes should be built on.

The same benefits are present when using aggregations, such as `sum`, `count`, `max`, etc. An aggregation typically involves performing a function on one or more columns. As with the conditions, the column-oriented structure will reduce the I/O since only the required columns will have to be read, while a row-oriented structure would need to read the entire table.

The benefits described above also introduce a tradeoff when it comes to random access of entire rows. Reading entire rows is more efficient if the row is consecutively stored. In a column-oriented database the parts of the tuple will be scattered over as many structures as there are columns.

In row-oriented databases the approach for increasing performance of predicate evaluation is to create indexes over these columns. The index maps values to tuple-ids and is ordered, thus allowing for binary search and for faster predicate evaluation. Creating indexes for all columns is hardly a good solution since the amount of space required explodes and also the bookkeeping costs of updates or inserts. The problem of determining which columns should be indexed can be handled using different built-in auto-tuning techniques or manually by a database administrator.

2.1.2 Compression

Higher compression ratios can generally be achieved for column-oriented databases than row-oriented databases since consecutive data is of uniform type in a column while data in a row is not[13]. Having data of uniform type opens up the possibility of using a wider selection of compression algorithms. For example run-length encoding (RLE), where the value and the number of times it repeats is stored as (value, run-length), could be a good approach for sorted columns where values are likely to repeat.

Column compression techniques can also reduce CPU usage by allowing operators such as aggregations to operate directly on the compressed data[5]. For example a `sum` aggregate function over run-length encoded data could easily perform the summation of each run-length tuple by first multiplying the value with the number of repeats. This reduces I/O significantly since the data is never decompressed.

2.1.3 Inserts, updates and deletes

The insert and delete operations are typical row operations, making them harder to handle efficiently in a column-oriented database. In a row-oriented database entire rows can be written in a single disk seek since it is consecutively stored. For column-oriented structures the different parts of the tuple is stored at different locations since each column is stored separately. This requires the tuples to be split and several writes must be performed. In column oriented databases the common approach is to try to buffer

inserts and then make bulk writes. This allows for writing all data to a specific column at once before proceeding to the next.

Another issue is the physical order of the data. Row-oriented databases typically use b-tree or b^+-trees files to achieve good insert performance for physically ordered files[8]. This amortizes the cost of re-ordering data over many inserts since the tree structure is only re-balanced when nodes become full. The average cost of the insert then becomes lower.

Some column-oriented databases, KDB for instance, keeps columns in entry order, only allowing inserts at the end of the column[2]. This makes inserts efficient, but there are obvious penalties to retrieval where some other ordering might have been preferable[15].

For deletes a typical approach for both types of databases is to use delete markers for tagging deleted rows and later purge the tagged rows in a batch operation. Updates can be handled as a delete followed by an insert.

2.2 Vectorization

Vectorization is an important concept in column-oriented databases. Compared to row-oriented databases where operations tend to work on a row-by-row basis, operations instead work on vectors of data of a specific column. This greatly reduces the amount of function calls needed and helps the compiler to identify optimizations and SIMD[1] capable code blocks. Many column-oriented databases are intended to run on commodity hardware, but today even these platforms have super-scalar processors with SIMD capabilities if the programs allow it. A binary vector operation is illustrated in Figure 2.1.

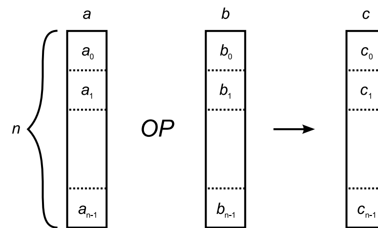


Figure 2.1: Illustration of a binary operator taking two vectors, a and b, as parameter and producing a vector c of the same length as result.

All basic operations are relatively easy to implement in a vector form, for example a comparison operator such as equality could be implemented using the C-style code in Listing 2.1. In this case the operator returns the indexes of the elements that were equal. The result could also be in form of a bit vector, where each bit represents one element. This is ideal for combining logical operators such as `AND`, `OR`, etc. since these can then be performed directly on the bit vectors that resulted from the operations.

This is similar to the Basic Linear Algebra Subprograms[1], BLAS, which is an interface standard for libraries to perform vector and matrix operations. The original implementation by Netlib is commonly used in high quality linear algebra software, such as LAPACK[3]. Large efforts has been made to make BLAS efficient and portable, thus it may also be useful to take advantage and learn from it. The vector based numerical

[1]Single Instruction Multiple Data, data level parallelism , for example SSE[12]

Listing 2.1: Vectorized equality operation.

```
int eq(int* vec1, int* vec2, int* result, int length)
{
    int j = 0;
    for(int i = 0; i < length; ++i)
    {
        if(vec1[i] == vec2[i])
            result[j++] = i;
    }
    return j;
}
```

operations in a column-oriented database could, no doubt, be based on some of the findings during the development of BLAS libraries.

Vector based operations are used in the MonetDB/X100 column-oriented database, developed at National Research Institute for Mathematics and Computer Science (CWI) in the Netherlands[16]. It takes advantage of modern super-scalar processors while also taking memory hierarchies and CPU caches into account. By working with vectors of lengths that will fit into L2 cache they can achieve even better performance with the vectorization. In the article the authors present performance gains of more than five times the performance of the original code after introducing vectorization and cache awareness. The primitive vector operations take advantage of the super-scalar processors achieving more than 2 instructions per cycle.

2.3 Insert, update and delete techniques

There exists arguments to why inserts should be slow in a column-oriented database:

1. tuples must be split into its different fields before insertion,
2. the fields must be stored at different locations, and
3. if a column is compressed and sorted, the overhead of an insert becomes even larger. This also applies to update and delete operations.

The easiest and least complex approach is to only insert data at the end of the columns, as is done in KDB and Addamark[15]. This allows for fast insertion but the tradeoff is a suboptimal read structure. Another approach is to try to buffer inserts and then do a bulk insert.

Deletes can basically be handled in two ways. The first and simplest way is to simply remove the fields from the corresponding columns and re-arrange the data to fill the gap. This is however costly and potentially implies locking large structures and reducing concurrency. The second approach is to keep some sort of delete index and simply add a delete marker into the index for each deleted tuple. At some later point the deleted records could be purged from the structure in a batch operation, which amortizes the locking/synchronization and latency costs over many delete queries. The disadvantage of this solution is the additional complexity to read operations.

Updates are easily handled if there is no physical ordering of the data. It is then simply a matter of finding the elements to update and overwrite them with the new values. Compression and sorted columns makes things more complicated however. The naïve, and slow, way to implement updates in that case is to decompress, update, sort,

compress and store the data again. If necessary for update performance the *sorted*-attribute on a column could be set to *false* to allow for in-place updates (when updates violate the sorting order).

2.3.1 Inserts, updates and deletes in C-store

C-store has another approach to handle inserts, updates and deletes. It uses two separate stores, the *read-optimized store* (RS) and the *writable store* (WS). The writable store is optimized for handling efficient inserts and updates. It does not use any encoding or compression schemes on the data since it is not expected to hold much data.

The much larger RS is optimized for read queries and is capable of handling very large amounts of data. Data is moved from the WS to RS by a *tuple-mover* which performs batch inserts into the RS. Only the tuple mover is allowed to manipulate data into the RS. Deletes are simply marked in the RS and later purged by the tuple mover. The RS compresses the data using run length encoding and bit vectors (see Section 2.4) depending on the type of column.

Updates are handled as a delete marker in the RS followed by a new insert into the WS.

2.4 Compression

As mentioned in Section 2.1.2 column-oriented databases allow for compression algorithms not applicable to row-oriented databases because of consecutive data being of uniform type. In most cases it is not minimizing the storage size that is the goal of the compression in column-oriented databases.

Analytical queries over large data sets typically involves looping over a few columns, performing a few operations per element. Using a column-oriented structure reduces the I/O significantly by only reading the columns of interest. The nature of the queries is still simple, in many cases the performance is still I/O bound[17][16]. This calls for light-weight compression algorithms with the goal to save I/O while not making CPU intensive queries suffer from high decompression costs.

Several techniques are suitable for this type of compression, one being the already mentioned run length encoding. In *Integrating compression and execution in column-oriented database systems* [5] Abadi, *et al.* evaluate a series of different compression schemes adapted to a column-oriented structure for the C-store database. Also in [13], [17] and [16] lightweight compression algorithms that were used in Sybase IQ and MonetDB are described. Below follows a summary of the techniques described by these papers:

– **Null supression -** In the paper by Abadi, *et al.* [5] a column-oriented implementation of null-suppression is described. This technique performs well for data sets with frequent nulls or zeros. It compresses the data by only using as many bytes necessary to represent the data. For example an integer type is typically 4 bytes, but small values may only need 1, 2 or 3 bytes to be represented. Two bits are used to tell how many bytes the field uses. In order to make the fields byte-aligned, the bits are combined for every four integers so that they occupy a whole byte. The representation is illustrated in Figure 2.2.

– **Dictionary encoding -** This type of compression is also widely used in row-oriented databases. The idea is to replace frequently occurring patterns with

smaller codes. For example if a column contains the names of cities that customers live in, the same names of cities are likely to exist at several locations. By mapping each string to a unique integer and storing the integers instead there is potentially a significant storage gain.

– **Bit vector encoding** - Bit vector encoding uses bit vectors to encode the values. One bit vector is required for each value, which makes it only useful when there is a limited amount of different values. For example the integer vector [1321311] would be represented by the three bit vectors b_1 : [1001011], b_2 : [0010000] and b_3 : [0100100].

– **Run length encoding** - Run length encoding is useful when values are likely to repeat. Instead of storing each repeated value, a tuple consisting of the value and its run length is stored. If a value is repeated n times, a run length encoding would reduce the storage space needed from a factor n to 1 plus the overhead of the run length field. The repeating values must be consecutively positioned for the run length encoding to work, as opposed to the dictionary encoding where the position of the values is irrelevant. This makes run length encoding particularly useful for sorted structures.

– **Frame of reference** - This type of compression uses a base value, a *frame of reference* and stores all values as offsets from the reference value. This method is useful when the values are not wide spread, since this will result in small offset values that can be represented by a fewer number of bits than the actual values. In [17] a version called *patched frame of reference* (PFOR) is used where a reference value is chosen for each disk block. It also introduces the concept of *exceptions* to handle the algorithm's vulnerability to values deviating a lot from the average. This reduces the range further, allowing for higher compression ratios.

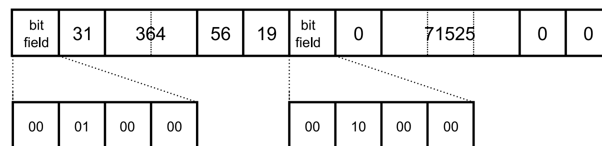


Figure 2.2: Null-suppression column structure. This example has eight integers and thus two bit fields telling how many bytes each integer field occupies.

2.4.1 Operating directly on compressed data

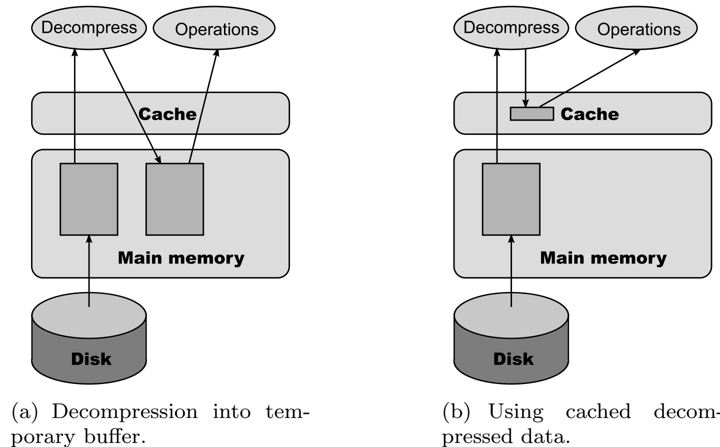
As previously mentioned in Section 2.1.2, some compression schemes allow for relatively easy operation directly on the compressed data. This clearly reduces I/O since the data need not be decompressed, but it can also reduce CPU requirements. In the paper by Abadi, *et al.*[5] the possibilities of integrating compression awareness into the query executor of C-Store[15] are investigated. The execution time benchmarks presented in the article shows that while normal compression trades I/O for CPU time, direct operation on compressed data can save both CPU time and I/O. The three compression methods evaluated are RLE, bit vector and dictionary encoding.

There is however one apparent tradeoff when implementing direct operations on compressed data, and that is implementation complexity. Basically, whenever a new

compression algorithm is added, all the implemented operations must also be made aware and be able to handle it. Also, the possibilities for operating directly on compressed data may vary with the type of operation to be executed.

2.4.2 Cache-aware decompression

A common approach when operating on compressed data is to decompress the data into a buffer in main memory and then operate on that buffer. This is illustrated in Figure 2.3(a). It is clear that the cache does not play an important role in this case since the data merely passes through it.



(a) Decompression into temporary buffer.

(b) Using cached decompressed data.

Figure 2.3: Decompression to cache vs. to RAM. Based on illustration in *Super-Scalar RAM-CPU Cache Compression*[17].

By instead pipelining the data from the decompression into the operation the data blocks are small enough to fit into the cache, and the intermediate buffer is no longer necessary. This is illustrated in Figure 2.3(b). Not only does this save I/O and utilize the cache better, but it also saves main memory storage space[17].

2.4.3 Selecting compression strategy

Selecting the appropriate compression technique involves taking many parameters into account. First of all is the type of the data. Many algorithms are only applicable to numeric data, such as FOR (or PFOR). Floating point values are typically high cardinality which calls for specific methods, etc.

Aspects that are harder to take into account are the values themselves and the patterns they are stored in. This includes typical range of the values, recurring patterns, sorting order, etc. It might be known beforehand and the administrator can explicitly choose the best compression scheme. In other cases some kind of automated method would be preferred. Typically some kind of analysis of the data is required for automation.

In C-store[15], four different encodings are used depending on the *ordering* (self or foreign) of the column, and the amount of distinct values it contains. Self ordering

means that the column is ordered by its own values, while foreign ordering means that it is ordered by corresponding values of another column.

1. *Self-order, low cardinality* columns are encoded using a type of run length encoding.
2. *Foreign-order, low cardinality* columns uses bit vector encoding as described earlier.
3. *Self-order, high cardinality* columns uses a type of frame of reference (FOR) encoding similar to the PFOR used in MonetDB.
4. *Foreign-order, high cardinality* columns are not encoded.

The expected workload is another aspect. If workloads are very CPU intensive, it might not even be appropriate to compress the data at all. If a query is CPU bound, adding compression might actually make the performance worse, unless CPU-cycles can be saved by directly operating on the compressed data as previously discussed. At least only lightweight schemes that allow for easy direct operation on the data should be considered.

If queries are I/O bound on the other hand, and light on the CPU, it might be justified to use a more expensive compression scheme. In the end it is about finding a balance between CPU usage and I/O to get the highest possible throughput.

2.5 Vertical partitioning vs. column-orientation

Vertical partitioning is a method for improving row-oriented database performance for *read-mostly* workloads. The method creates n new tables from a table with n columns[6]. Table 2.1 would result in the three tables Table 2.2(a), Table 2.2(b), Table 2.2(c).

Table 2.2: Result from vertical partitioning of Table 2.1.

(a) The id column

tid	id
0	524
1	525
2	530

(b) The first name column

tid	fname
0	John
1	Jane
2	Bill

(c) The last name column

tid	lname
0	Smith
1	Grant
2	Johnson

Creating a new table for *all* columns is the extreme case of vertical partitioning, taking the row-oriented table very close to a column-oriented structure. Creating many tables will result in more storage overhead, not only because of the additional column for each table (*tid*) but also because of a tuple overhead that is present for all tuples[6]. Because of this, vertical partitioning is in reality only used for the columns that are expected to be read most frequently.

In *Column-stores vs. row-stores: how different are they really?*[6], Abadi *et al.* sets up a benchmark to compare vertical partitioning performance in a commercial row-oriented DBMS with the performance achieved in a column-oriented database. The conclusion is that vertical partitioning could not improve the performance when performed as described above. Further the authors note that the performance is not only

a magnitude worse than the column-oriented database, but also worse than the performance of the original row-oriented database without vertical partitioning. In a web log post the same authors claim that the following reasons are likely causes to the result[4]:

1. **Size overhead** - The additional size introduced by the tuple-id column along with tuple overheads produce too much overhead. The header and tuple-id together are much larger than the actual data in many cases. In the column-oriented database this is not the case and I/O can actually be improved. For the vertical partitioning however, this results in an increased amount of I/O in some cases.

2. **No horizontal partitioning** - The row-store product used was no longer able to take advantage of horizontal partitioning, potentially imposing a performance penalty.

3. **Joining partitions** - Joining the partitions on tuple-id is a fast operation since the tables are clustered on that column. The authors found that it was the large amount of joins that occasionally caused the query optimizer to execute suboptimal plans.

The bottom line from the article is that column-oriented databases has a significant advantage from the fact that code throughout the whole system, including query planner, query executer, storage handling, etc., is optimized for column structured data. Row-oriented databases will always have to make tradeoffs to perform well for both column-oriented and row-oriented workloads. It should be noted, however, that the authors had difficulties with making the row-oriented database understand the vertical partitioning and that the results could be improved if a few features had been added to the product.

Chapter 3

Java performance

Java is an object oriented cross-platform language. Applications are typically compiled to bytecode which can be executed by a Java virtual machine (JVM). The major benefit is that the same code can be run on any platform and regardless of the computer architecture. Run-time interpretation of bytecode does introduce overhead however, and it will almost always result in slower execution than code compiled into native executables. Recent JVM developments has tried to narrow the gap by providing a series of optimization techniques. One of these techniques is *just-in-time* compilation which compiles select parts bytecode into native code when it is executed, and then caches it. This narrows the performance gap between native executables and bytecode, but there is still an overhead associated with compiling during execution.

Java features a simple object model compared to, for example, C++. It does not support multiple inheritance and the memory management is automatically handled by the garbage collector. In Java, all built-in classes and user defined classes are implicitly derived from the `Object` class, either directly or indirectly through some other class. The intrinsic types (primitive types) are the only exceptions that are not classes. The primitive types were kept because of performance and memory reasons. Generic programming capabilities were introduced in Java 1.5 and it is called *generics*. It only allows for type parameterization of non-primitive types however. Thus, the whole API built around generics is also limited to using objects. This has implications both to memory usage and performance for generic implementations.

Both memory overhead and poor performance are detrimental to a database system. For this reason a few things has to be investigated before a Java-based column-oriented database implementation can be designed. The evaluation is based on benchmarks and tests performed with Sun's HotSpot VM 1.6.0_10. The reason this virtual machine is used is that it is the most widespread and popular implementation of the Java Virtual Machine Specification, at least when taking earlier versions of it into account as well.

3.1 Memory overhead

The memory overhead in Java is mostly due to the `Object` base class that all non-primitives inherit. 8-byte alignment results in that an object of type `Integer` takes a whole 16 bytes of memory, compared to 4 bytes for the primitive integer type.

In a small test program from an article by Vladimir Roubstov[14] the size of different Java data types is calculated. This program was used on Sun's HotSpot VM 1.6.0_10 for

Windows to get the following results. A primitive `int`-array of length 100 was compared to an `ArrayList<Integer>` with the same capacity. The program calculates the memory consumption by comparing the size of the heap before and after allocation.

The `int[100]` array requires 416 bytes of heap space which is about what one would expect. 100×4 bytes $= 400$ bytes, plus 16 bytes for the array object.

An `ArrayList<Integer>` with capacity of 100 takes 440 bytes, which is not very different. The problem however is that this list does not yet store any values, it simply has 100 references to `null`. Creating and inserting 100 `Integer` objects into this list will require an additional 16×100 bytes, a total of $= 1600$ bytes. Hence, the total memory usage using an `ArrayList<Integer>` is 2040 bytes. So why not just use the primitive types one might ask.

The problem with using primitive types in Java is that these types can not be parameterized with generics. The result of this is that the whole Java Collections API can not be used with primitive types either. Storing primitive types in arrays keeps memory overheads low, but the tradeoff is that type parameterization can not be used.

Auto-boxing was introduced in Java 1.5. It makes it possible to use primitive type arguments to methods that take the corresponding object type parameter. For example passing an `int` argument to a method that takes an `Integer` parameter. This might trick some to believe that perhaps the data stored in collections might be in its primitive form as well, but this is not the case. What really happens is that the auto-boxing feature automatically, and transparently to the user, wraps the primitive type into an object and then passes the object reference to the method instead. In terms of memory overhead this is not any improvement compared to manually constructing the object and passing its reference. The auto-boxing features a cache however, which guarantees that auto-boxing for integral values in the range $[-128, 127]$ returns the same object[11], so for values in that domain space can be saved. For applications where values are not typically in that range however, the large memory overhead associated with storing objects remains.

3.1.1 Java strings

Using the same program the minimum size of a `String` can be calculated to 40 bytes, which is quite alarming considering that it is enough memory to fit 20 Java `char` values. As the strings grow however, the additional space consumption is only 2 bytes per character, allocated 8 bytes at a time. For small strings on the other hand, the 24 bytes of extra overhead (16 bytes is for the internal `char`-array) is very large relative to the size of the amount of useful data. Table 3.1 shows the size of strings and the relative size of the overhead. For the empty string, the relative overhead is $\frac{(40-2)}{2}$ making a total of 1900%. For larger strings the relative overhead decreases, but many applications typically use strings of length 0 to 10.

There is, however, an optimization in some Java implementations to save memory for Strings and that is string caching. For example in the code in Listing 3.1 both `foo1` and `foo2` could reference the same object, hence the second string will not require any additional memory. The reason that this can be done is because the `String` class is immutable. Note that the String caching features are implementation specific and should not be taken for granted.

The alternative is to represent strings in some other way. Using a `char`-array for storing the characters is one option. This requires 16 bytes for the array object which is still a lot of overhead for small strings, but also a notable improvement over the `String`

Table 3.1: Memory consumption of Java `String`.

Length	Size(bytes)	Relative overhead
0	40	1900%
1	40	900%
2	40	566%
3	48	500%
4	48	380%
5	48	300%
6	48	243%
7	56	250%
8	56	211%
9	56	180%
10	56	155%
100	240	18%

Listing 3.1: String caching example.

```
String foo1 = new String("foo");
String foo2 = new String("foo");

/*Will evaluate to true if foo was cached, i.e. foo2 references the
  same object as foo1 */
foo1 == foo2;
```

class. The `char` array also integrates well with Java and can easily be wrapped by `String` objects when necessary.

3.2 Loops over primitive types

In Section 2.2 and Listing 2.1 the benefits of operating on vectors are discussed, the most important being less method calls and better use of modern CPU features. It is also relevant to investigate how Java performs for such operations. A small benchmark was written to compare performance of a Java program to an equivalent program written in C++. The benchmark uses similar operations to those in Listing 2.1.

The results of the benchmark showed that Java performed at least as well as the C++ program, which is very good news for this application. Many extra flags had to be supplied to the C++ compiler for it to match the performance of Sun's HotSpot VM. The results are displayed in Figure 3.1. The benchmark used three simple operations: subtraction, division and a comparison. An 8-iteration loop-unrolled version of each was also tested but did not affect performance much since both compilers were capable of loop unrolling automatically. Compiler settings and the complete results are presented in full in Appendix B.2.

As mentioned earlier, as of Java HotSpot VM 1.3 loop unrolling, instruction scheduling and range check elimination are built in features into the VM[10].

The reason this benchmark is important is that simple operations such as comparisons are vital parts of the performance in a column oriented database. Typical queries generally involve some simple predicate to select a few rows from a large table. It is then the amount, not the complexity, of operations that decides the performance.

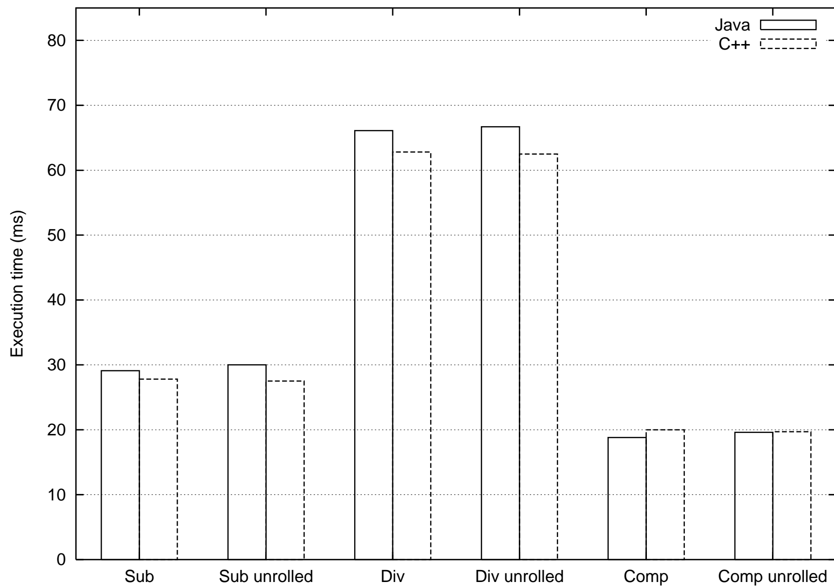


Figure 3.1: Benchmark results for operations on primitive type arrays.

3.3 Conclusions

It is quite clear that objects should be avoided when it comes to storing large amounts of data in Java. Storing data using the primitive types can be done with very little overhead, and the benchmarks show that performance over primitive type arrays is very good.

When it comes to storing strings it is hard to make much improvement. Storing the string as a `char`-array still adds a 16 byte overhead for the array object. Caching strings can reduce the memory usage quite a bit, especially for small domains.

Chapter 4

Implementation

The design of Coldbase is derived from the study of existing column-oriented database products and the methods used in the area (see Section 2). It also uses the methods described in Section 1.2.4. As this implementation is part of a thesis project it should be considered to be a proof-of-concept application. As stated in Section 1.2.2, the goal is to implement a design with the fundamental functionality necessary to be able to evaluate the performance while also keeping the interface easy to use and extend.

This chapter describes the different parts of the design in a bottom-up fashion. It begins by describing the core data types and the interface for manipulating vectors. Then the concurrency control structures are explained. Finally the query interface built on top of the system is described.

4.1 Data types and representations

From the Java memory review in Section 3.1 it is clear that data should be stored using arrays of primitive types necessary, both from a memory overhead point of view and a performance point of view. The downside was also pointed out: no type parameterization can be used and method dispatching is not possible. The consequence is that the basic data types must be implemented separately, even if they have many similar functionality. For example operators for floating point types and integral types are similar or identical, but these methods must still be duplicated because the underlying array-type is different.

4.1.1 Vectors

All data is stored in vectors. The base class for the vectors is the `AbstractVector` class which provides the common interface for manipulating vectors. The inheritance graph for vectors is illustrated in Figure 4.1.

The common interface for all vectors includes methods for getting, appending, updating, comparing(equals) to other vectors and values and searching for data. Most of these methods are vector methods in the sense that they either return a vector as result or takes a vector as argument. For example the update method performs multiple updates in one call on the specified indexes and with the specified vector of new values.

The most important methods are the indexing methods which allows for retrieving specific elements from the vector. Indexing is done by specifying the indexes of the

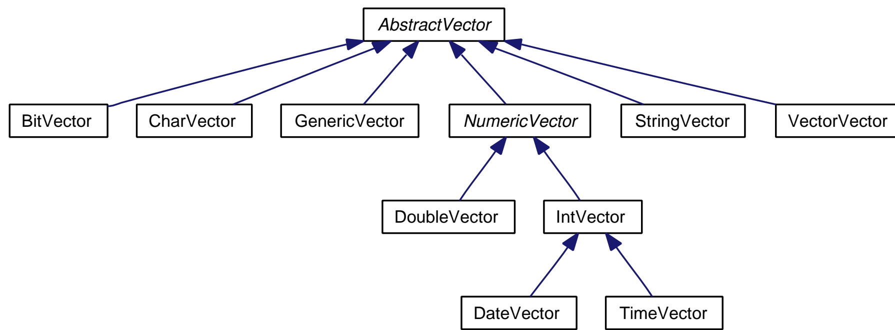


Figure 4.1: Inheritance graph for vector types, abstract classes are *italic*.

wanted elements. The method returns a vector of the same length as the indexing vector. Thus if v_i is a vector with indexes and it is used to index a vector v_1 producing the result vector v_2, $v_2 = v_1[v_i]$, then $|v_2| = |v_i|$. The returned vector v_2 is of the same type as v_1. More details about indexing can be found in Section 4.1.8.

The `AbstractVector` is an abstract class and it does not define how data should be stored. This is defined by its subclasses.

4.1.2 Atoms

An *atom* is a special vector that only contains one element and can be used both as a vector of arbitrary length or as a single value. All types of vectors can be atoms. If an operation is performed on an atom that requires it to act as a vector it will automatically be handled as a vector instead, where each element is equal to the atom value. For example if one atom has the integer value 10 and it is multiplied by a vector $[1, 3, 5]$ the result will be $[10, 30, 50]$. The interface is mostly vector based which could remove the possibility to perform operations with single values. Atoms makes it possible to use single values transparently on the same interface by automatically expanding into the same length as the other operand.

Coercion of atoms into vectors is handled through a protected method in the vector interface. The method takes another vector as parameter and expands the atom into the same length as the parameter vector. By calling this method, vector operations can expand the atom into a vector of the same length and then perform the same element-wise as it usually would. An example of this using numeric vectors is given in Section 4.1.3 and illustrated in Figure 4.2.

The vector types have a constructor which can take a single value and will then create an atom of the corresponding type. Appending more values to the vector afterwards will remove the *atom* attribute and it can then only be treated as a vector. Vector coercion of a vector where the atom attribute is not set will not have any effect. A vector can also be turned into an atom manually if it only has one element by calling the `makeAtom`-method. This can be useful if the vector after subsequent operations is left with one element but with the *atom* attribute removed.

4.1.3 Numeric types

`NumericVector` (see Figure 4.1) is the base class for vectors containing numeric types. It implements the `Comparable`-interface, which declares the comparison methods. Comparison methods include typical ordering comparisons such as *greater than*, *less than*, etc. but also equality, inequality, *min*, *max* and intersection. The numeric vector also provides an interface for the numeric operations such as addition, subtraction, multiplication etc.

The `IntVector` and `DoubleVector` implements the interface for integer and double types. Both of these classes use an array of the corresponding primitive type for storage. The underlying array grows by calling a growth-function. The current implementation uses the following function: $newsize = size \times 2 + 16$. This makes growth a bit faster for small arrays and for larger arrays the new size is about twice the old size. When this happens, a new array of the new size is allocated and the data from the old array is copied.

The typical operations on the numeric vectors are element-by-element operations and the result is a vector. The binary operators in the `Comparable`-interface are boolean operations, hence the return type is a vector of booleans (see bit vectors in Section 4.1.9). For numeric operations such as multiplication, addition, etc. the return type is numeric vector. As the operations are element-wise the resulting vector is of the same length as the operands.

Generally when performing numerical operations on vectors it is also desirable to be able to use a constant. For example if one wishes to add the number 10 to all elements in a vector it would be convenient if the interface allowed for it directly. This can be done using the atoms described in the previous section. If an atom is one operand in the operation it will automatically be expanded to the length of the other operand if used in a vector operation by using the method described in Section 4.1.2. The result is that for binary vector operations both operands must be of the same length, or one of the operands must be an atom. Figure 4.2 illustrates a *greater than or equal to* comparison between a vector and an atom. The intermediate step expands the atom to the length of the other operand and then the comparison is made element-wise, resulting in a bit vector.

Figure 4.2: Example illustrating a *greater than or equal to* comparison between a vector and an atom, resulting in a bit vector. First the atom is automatically expanded to the length of the other operand and then the comparison is made.

Vectors storing numeric data also have an internal *sorted* attribute, which determines if the vector is sorted or not. The attribute is maintained during update and append operations, possibly removing it if new data violates the sorting order. Maintaining the

Listing 4.1: Example vector operation.

```
IntVector v1 = new IntVector(new int[]{1,2,3,10,9,8}, false);
DoubleVector v2 = new DoubleVector(new double[]{6.3, 2.6, 7.1, 7.3,
  1.6, 99.9}, false);

/* v1 coerces into double before multiplication, but after the
   operation v1 is unchanged. The result is of type DOUBLE. The
   interface is specified in the NumericVector-class, thus the
   result must be cast into the appropriate type.
   */
DoubleVector result =(DoubleVector) v1.mul(v2);

/* Also when v1 is the parameter of the operation will it be
   coerced into DOUBLE before the operation takes place. */
DoubleVector result2 = (DoubleVector)v2.sub(v1);
```

attribute is relatively cheap considering the gains that can be made when searching the vector. The find operation for example is possible to do in $O(\log n)$ using binary search instead of $O(n)$. Some operations only keep the attribute when it can be done at a low cost, for example the add operation will set the sorted attribute only if both operands are sorted, which can be checked very easily.

4.1.4 Type conversions and coercions

Numeric types will automatically be coerced when necessary and the interface for coercion is defined in the `NumericVector` class. Coercion is based on the order of the numeric types in the `DataType` enumeration. Currently there are only two basic numeric types implemented: `INT` and `DOUBLE`, but if more types were implemented their order would be like in Table 4.1

Table 4.1: Coercion order of numeric types.

Order	Type
1	SHORT
2	INT
3	LONG
4	FLOAT
5	DOUBLE

Let the coercion order of a vector v be given by the function $C(v)$ and the type be given by the function $T(v)$. When a comparison or numeric operation is performed on two numeric vectors, v_1 and v_2, their coercion order is first compared. If $C(v_1) < C(v_2)$ then v_1 is converted to the type $T(v_2)$, else if $C(v_2) < C(v_1)$ then v_2 is converted to the type $T(v_1)$. Both vectors are afterwards in same the type as the operand with highest coercion order. When the coercion is made the operation or comparison is executed.

Note that the type conversion is only made on the temporary vector that is used to execute the operation and does not affect the input vector. Example code where automatic type coercion takes place is displayed in Listing 4.1

4.1.5 Date and time

As one might guess from the inheritance graph in Figure 4.1, dates and times are stored as integer numbers. Dates are stored as integers representing the number of days since 1970-01-01 and times as integers representing the number of milliseconds since 00:00:00.000.

The fact that they are derived from the `IntVector` means that they also inherit all the operators that can be used on numerical types. This allows for comparing dates/-times, adding or subtracting from dates/times, etc. The constructors of these classes provide convenient parsing of dates and times so that atoms can easily be created from a string representation of a date or time. An example of typical manipulation of dates can be found in Listing A.2.

4.1.6 Characters and Strings

`CharVector` stores characters as an array of `char` and it employes the same growth function as for numeric types. The vector is primarily used to represent physical strings.

The type `StringVector` in Coldbase on the other hand is stores strings in a completely internalized fashion. An internalization map maps each string to an integer which is the index of the stored string in the internalization table. The integer then works as the reference to the string. In the `StringVector` only the integer references are stored in a `int`-array, thus each added string only takes 4 additional bytes if it already cached. For the physical storage of strings in the internalization table, the `CharVector` is used. This type of caching makes new instances of already existing strings only take 4 bytes extra, even if they are in different vectors or even different tables in the database.

Since each integer references a unique string, equality comparisons can be done directly on the integers without needing to look up the actual strings (by simply comparing the integers). This makes string equality operations very efficient. In typical queries the equality operation is perhaps the most frequently used operation, so this effect of internalization is very attractive. Other type of operations requires looking up the string in the table and doing the operation on the actual string.

The interface of the `StringVector` does not automatically look up the corresponding strings when elements are indexed, but simply returns the integer references to them. The user must then use the internalization table to look up the real strings. The reason for this is to make sure no unnecessary performance penalties are made by looking up the strings when the user might actually only need the references. The query system however makes the underlying integer references more transparent to the user. More on this in Section 4.4 about the query system.

The string vector also implements the `Comparable` interface which makes it possible to compare strings lexicographically and for equality as described earlier. The *greater than*, *less than*, etc. comparison operators can not operate directly on the integers since the value of the integers has nothing to do with the lexicographical order of the strings. The strings must instead be looked up before the comparison can be made.

There is one main reason that string internalization is necessary when some Java implementations already cache strings, and that is control. Providing the caching facilities in the implementation guarantees control over the physical storage of the strings. For example serializing the internalization table is not a big issue. This way the internalization table can be restored at a later time and the references stored in tables (the integers) will still be valid. Accessing the strings cached internally by the virtual

machine is harder however. And even if they were serialized and could be restored at a later point the stored references in the tables would be invalid.

4.1.7 Generic types

To allow for generic types and vector types (vectors of vectors) the `GenericVector` and `VectorVector` have been implemented. These use Java collections as underlying storage and do not perform as well as the other types. Other vector types get their low memory overhead and good performance from storing data in primitive type arrays. This is not possible for generic types however, which must be stored as objects. This means more memory overhead and less performance because of additional dereferencing before the actual data can be accessed.

The generic vector types are implemented to support generic and user defined types. The user should be aware that performance when using generic types is not as good as the primitive type vectors.

4.1.8 Indexing and finding

Most methods in the interface for vectors operate over the vector, typically taking another vector as parameter and performing some type of element-wise operation and returning a third, new, vector.

Methods that involve finding values or comparing always return indexes. To get the implied element in the vector the `index` method must be called with the index values that the operation provided. For example `find`, which can be used to search for a single value or a vector of values, returns the index of the first occurrence of each sought value. If the value is not found the invalid index (-1) is returned. For example searching for $[1, 3, 5]$ in the vector $[5, 5, 9, 3, 7]$ will result in the index vector $[-1, 3, 0]$. Notice that the returned vector is always of the same length as the vector of sought values.

If an indexing method is called with an index that is outside the range of the vector an exception will be thrown. There is however a special method called `indexNulls` which allows for out of range indexes. This method instead returns a null value (see *Null value representations* in Section 4.1.10) for those elements.

The `Comparable` interface uses another type of indexing, bit vectors. The comparison operators are boolean operators that operate over vectors element-wise. One comparison is made for each element resulting in one boolean value, or one bit, per element. Thus the result of a comparison is a bit vector of the same length as the operands, where each bit represents the result the comparison of the elements of corresponding index. For example comparing $[1, 2, 4, 6, 9]$ with $[1, 2, 5, 7, 9]$ for equality results in the bit vector $[1, 1, 0, 0, 1]$, where set bits represent the elements that were equal and the zero bits represents elements that were not equal.

Using bit vector indexes instead of ordinary integer indexes for this type of operation has its advantages and disadvantages. The disadvantage is that the bit vector will typically have to be converted into an integer index vector when the values have must be fetched. The biggest advantage is that logical operators can easily be used with the bit vectors. Consider the following query: *all integers equal to 10 or less than 2.* This could easily done by performing both comparisons, resulting in two bit vectors, and then performing a bit-wise OR operation on the bit vectors (which is a very fast operation, see Section 4.1.9). Size might be another advantage, depending on how sparsely the comparison is evaluated to *true*. An integer index will require 32 bits for each set bit

(if 32-bit integers are used), but it does not require any space at all for zero bits. Bit vectors on the other hand requires only one bit for each comparison evaluating to *true*, but also *false* takes one bit. This results in that if a comparison is evaluates to *true* in less than 1 case out of 32 an integer index would be more space effective, but otherwise a bit index is more effective. The advantage of fast logical operations made the bit vector a more attractive solution.

4.1.9 Bit vectors

Bit vectors use a `long` array to store the bits. This allows for fast bitwise operations that can operate on a whole `long` at the time. Other operations become a bit more complex however. Indexing first involves determining the correct `long` in the array, and then shifting out the correct bit.

The most important method is perhaps the `where` method which translates a bit vector into an integer index vector. For example the vector from Section 4.1.8: $[1, 1, 0, 0, 1]$, this vector represented whether the elements passed or did not pass the equality condition. Translating this into an index array would result in $[0, 1, 4]$, i.e. the indexes of the set bits. The `where` method uses a `switch` statement that operates on four bits at the time. This requires 16 iterations per `long` and gives decent performance. The source code for the `where` method is attached in Appendix C.1.

An internal counter keeps track of how many bits are used in total. A growth function similar to the one used for numeric types (see Section 4.1.3) is used for allocating more `long`s.

4.1.10 Null value representations

Null values are represented for each primitive type as a specific value of the type. Table 4.2 shows the values representing null for each type.

Table 4.2: Null values for primitive types.

Type	Null Value
Integer	-2147483648
Double	*NaN*
String	0 or `"null"`
Char	0
Time	-2147483648
Date	-2147483648
Bit	0
Generic	`null`
Vector	`null`

Using a specific value for null does not require any additional structures and does not increase the implementation complexity much. The disadvantage is that some types are hard to find a suitable null value for.

Integers use the minimum value as null. Doubles use *NaN* for null, as defined by the Java specification of the `Double` type. Since the string type is internalized and represented as an integer the number 0 is reserved for null values. This is done by inserting `"null"` into the string internalization table upon initialization which will map 0 to `"null"`. Date and time types are based on the integer type and thus inherits its

null-value as well. The small range of bits is a problem, the 0 bit represents null. Generic types and vector types store references and can use the `null` reference as null value.

4.2 Map

At first the built-in Java `java.util.HashMap` for the index map (used in the algorithm for group-by) and key map (used in the construction of the left join, see Section 4.3.1). This resulted in horrible performance however, mostly because it was not possible to use the vector based operations defined by the vector interface. Using the Java implementation restricted the algorithm to work on a row-by-row basis for inserting and getting values, which resulted in many more method calls. A new vector-based implementation needed to be designed.

Constructing a vector-based hash-map is not completely trivial however. The first step includes vector hashing of all types, hence every data type now implements a fast `hash`-function that returns an integer vector with the hashes for every element in the vector. The hash function depends on the data type, for example integers return a reference to the vector itself and therefore is extremely fast, floating point types use the same function as Java, etc.

The following example explains how an insert of three columns that will act as keys for the map is performed. The reader should be well acquainted with how hash-maps work to understand this implementation. The first step is to create the hash vectors, combine into a single hash vector and create the corresponding index vector. The index vector represents the indexes where the values should be stored in the map. Combining the hash vectors into a single hash vector is done by multiplying with a large prime number p: $h = p * h + h_{other}$ where h is the result hash and h_{other} is the hash that is combined with it. Since h and h_{other} are vectors in this case, this will also be vector operations. For three columns this becomes $h = p * (p * h_1 + h_2) + h_3$ where multiplications and additions are element-wise vector operations. Calculating the index is a vector modulo operation with the size of the map. The first step is illustrated in Figure 4.3.

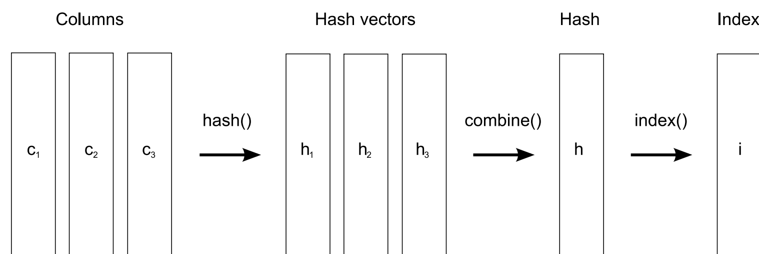


Figure 4.3: First step of inserting into the map, including hashing and creating the index vector.

Next the index vector is used to insert the values into the map. If the given index points to a free spot in the map, the value is simply inserted. If there already is a value at the position an equality comparison must be done in order to determine if it is a match. The insert method collects all the values that must be compared in a vector and returns it.

Now the values that were not inserted must be compared with the values already

present in the map. This is again performed as a vector operation. First the actual values are fetched from the map and put into a vector and then the element-wise comparison takes place. This comparison returns a bit vector, where set bits represents the values that were equal. The equal values can be inserted, appended if it is an index-map or replacing old values if it is a key-map. To get the indexes of the values that were not equal, the bit vector is flipped using a `not`-operation. Now the whole insert-procedure is repeated with an offset on the insert.

Once the insert method does not return any values, all values have successfully been inserted into the map. The algorithm could be summarized in short as:

1. Hash key columns
2. Combine hashes
3. Calculate corresponding positions in map, the index vector $index$
4. Insert values v using index vector $index$ at $offset = 0$
   (a) For every remaining value $v[i]$, try inserting into $map[index[i] + offset]$
   (b) Collect and return values v and the indexes $index$ of the values that could not be inserted
5. Compare v with $map[index]$ (v and $map[index]$ are both vectors)
6. For those that were equal, insert into map (update or append)
7. For those that were not equal, assign to v and corresponding indexes to $index$ and go to step 4 with $offset = offset + 1$

This map uses the primitive operations defined by the vector interface and can benefit from operating on the whole vectors at once instead of iterating row by row. This gives it a performance advantage compared to using e.g. the Java `HashMap`. Benchmarks comparing an implementation based on the Java `HashMap` and the implemented maps are attached in Appendix B.3. The benchmarks show that the key map is significantly faster than a corresponding implementation based on the `HashMap` while the index map is slightly faster.

Getting values from the map is also vector based and pretty much follows the same algorithm as inserts. The sought keys are hashed, combined and fetched from the map, followed by a comparison. The same type of looping is required for increasing offsets. This makes lookups vector based and gives very good performance. The lookup performance is critical for operations such as the `loj`, the left outer join (see Section 4.3.1). The good performance of the lookup is evident in the join benchmark in Appendix B.4.4.

The goal is of course to require as few iterations in the loop as possible. To achieve this a large map should be used initially and the hash function must not have a high periodicity. Using a prime number as size of the map is also a defense against poor quality hash functions. The size of the map is therefore selected from a set of prime numbers. To support further inserts into the map it must also be allowed to grow. This requires rehashing of the whole map. In the case for group-by operations the map is only inserted into once when the grouping is made, so rehashing and growing was only implemented for key-maps.

4.3 Tables

Tables are defined by a collection of vectors, which logically become the columns of the table. They also store the names of the columns, which are mapped to their corresponding vector.

The interface declares methods for row-wise operations such as appends and updates. The methods allows for appending or updating several rows at once by using the vector methods in the underlying vectors. For example the `bulkAppend` method takes vectors as parameter, where the first vector will be appended to the first column, the second vector will be appended to the second column, etc. This utilizes the efficient vector based interface of the underlying vectors.

Synchronization and concurrency control is automatically provided by the table. When appends or updates are performed the table is implicitly locked for writing or appending, making the thread wait if the lock is currently not available. Methods that retrieves data locks the table for reading. The lock can also be retrieved to use manual external synchronization, which can be useful if direct manipulation to the underlying vectors is required. For more details on synchronization and concurrency control, see Section 4.5

Tables can also have keys over one or more columns, creating a uniqueness constraint on the key-part of the tuples. If a table is keyed, the behavior of some methods is altered. The append and insert methods will act as updates instead if the key-part of the new row already exists. The method will then update the existing row with the new value instead.

Keys are implemented using a vector based map described in Section 4.2. The map used for keys has a single value for each key which gives the update behavior for duplicates. The table can also be keyed when it already contains data. This will act similarly to inserting all the data into a new keyed tabled, resulting in that all duplicates are removed. In the case of duplicates, the tuples with highest index will remain since they are inserted in index order.

4.3.1 Left outer join

Left outer join can be performed on two tables, in which each row of one table, the *source*-table, is coupled with all matching rows of the other table, the *lookup*-table. The lookup table should be a keyed table and the join is performed over the columns matching the key columns in the source table. In the resulting table there is at least one row for each row in the source, possibly with null values if the key did not exist in the lookup table. An example of a left outer join operation exists in Table 4.3. In this example the transactions of products is joined with a table of products. To make column names unique a suffix *(n)* is added to the column name.

The left join is simply a lookup in the key map of the lookup-table, hence the name *lookup*. The lookup will return the rows for the keys that were present, or null for non-existing keys.

In Coldbase the left join is done by using the key map (see Section 4.2) from the lookup table. First of all, it implicitly locks both tables for reading. The columns in the source table corresponding to the key columns in the lookup table are used as parameter to the `get` method of the map, which performs a tuple-wise lookup. Since the key map offers a vector based lookup method this operation is very fast. The result is a vector with the indexes of sought keys in the lookup table, or -1 for the keys that

Table 4.3: Example of left outer join, *source* `loj` *lookup*.

(a) Table with transactions of products, acting as *source* in the join. The **pid** column is a foreign key into the product table.

tid	pid	count
0	2	5
1	1	10
2	3	3
3	2	1
4	0	5
5	6	0

(b) Table with products, acting as *lookup* in the join. The **pid** column is primary key.

pid	pname
0	Super Chocolate
1	Chew me!
2	Banana roll
3	Sour fizz

(c) Result of join. No product was found in *lookup* for **pid**= 5 producing null values for that row.

tid	pid	count	pid(1)	pname
0	2	5	2	Banana roll
1	1	10	1	Chew me!
2	3	3	3	Sour fizz
3	2	1	2	Banana roll
4	0	5	0	Super Chocolate
5	6	0	null	null

were not found. This index vector is then used to retrieve the values of the columns in the lookup table. Since the index vector contains -1 indexes the `indexNulls` method is used (described in Section 4.1.8) which will give those rows null values. The names of the joined columns are appended the suffix to make them unique.

When the join is complete the locks are released and the resulting joined table is returned.

4.4 Queries

Apart from performance, the other goals (Section 1.2.2) were to keep a clean and extendable interface. The interface implemented at vector level is supposed to implement the core operations that operate directly on the primitive type arrays. The goal is to provide enough primitive operations to be able to construct most other operations in terms of combinations of the primitive ones.

The query interface is built on top of the vectors, not manipulating the primitive arrays directly but simply using the operations in the vector interfaces. Queries are built by composing different objects in a *composite* pattern[9]. There are some similarities to SQL in how the queries can be composed and the naming of the different classes. As an example a simple SELECT query is displayed in Listing 4.2. This query will get the column *col* from the table *table* with no conditions or groupings applied. It would correspond to the SQL query `SELECT col FROM table`.

The `Select` class is derived from `AbstractQuery<AbstractTable>` indicating that it returns a table upon execution. Using this base class for queries allows for query

Listing 4.2: Example query.

```
AbstractTable result;
result = new Select(new Column[]{new Column("col")}, table, null, null);
```

Listing 4.3: Example query with condition. Selects **name** and **age** from *table* where **age** ≤ 50.

```
AbstractTable result;
result = new Select(new Column[]{new Column("name"), new Column("age")}, table,
                    new Condition[]{
                        new Comparison(new Column("age"),Comparison.Type.LEQ,new Atom(50))
                    }, null);
```

nesting. For example a select might have another query as input, as long as it specifies a compatible return type.

Figure 4.4 illustrates the inheritance graph for the objects in the query system. The common method inherited from the `Column` class is `execute`, which always returns a vector with the result from an operation. The base operation defined by `Column` simply returns the specified column, as in the example in Listing 4.2. All derived classes should be able to take other `Column`-type objects as input, so that operations can be nested.

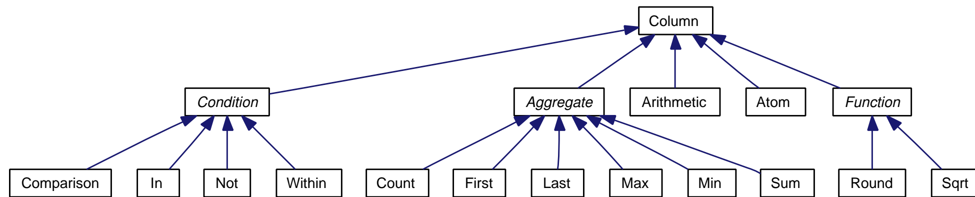


Figure 4.4: Inheritance graph for the query system, abstract classes are *italic*.

Every object is responsible for checking and throwing the appropriate exception if it can not operate on the data type of the supplied column, or if it does not exist in the source table.

4.4.1 Conditions

The condition sub branch in Figure 4.4 include the implemented conditional query objects. `Comparison` implements the comparison operators greater-than, less-than, equal, etc., `Within` implements a range condition and so on. The condition objects typically use the `Comparable` interface of the vectors and will result in a bit vector upon execution which is typically used by a select query to filter out rows, but it can also be a part of the result.

The `Not` condition typically operates on a nested condition and simply returns a bit-wise flipped bit vector.

4.4.2 Aggregates

Aggregate functions generally operate over a column and return a single value, unless groupings are present. If a single value is returned, this is in form of an atom vector(see Section 4.1.2). If other operations in the query return more than one value, the atom vector will be expanded to a vector with the same length as the rest of the result table. For example the following query in pseudo-SQL `SELECT sum(pid), pname from table` performed on Table 4.3(b) would result in Table 4.4. The sum produces an atom value 6, but it is expanded to match the length of the **pname** column when the table is produced.

Table 4.4: Result of `SELECT sum(pid), pname from Table 4.3(b)` illustrating aggregates and atom expansion. The sum of the **pid** column is an atom value but it is expanded to the length of the **pname** column.

sum(pid)	pname
6	Super Chocolate
6	Chew me!
6	Banana roll
6	Sour fizz

If aggregates are performed over a grouped results, one value per group is returned instead. More on how groupings are performed is discussed in Section 4.4.3.

4.4.3 Group-by

The group-by part of a select query is the last argument. Group-by generates a grouped result similar to the GROUP BY clause in SQL. Groupings can be performed on one or more columns and all tuples with matching values for the grouped columns will be contained in a specific group. When aggregate functions are executed these are performed over each group.

To get a clearer picture of how a GROUP BY query works, consider Table 4.5(a) with persons living in cities and their ages. Now consider the query `SELECT city, max(age) from table GROUP BY city`. This query will group the table by **city**, creating three groups (one for each city). `max` is an aggregate function which will be performed on each group. For New York, the oldest person is Paul, 53 years old. For the second group, Chicago, the oldest person is Bill, 30 years, etc. The `city` column is simply selected from the groups producing the three city names. The query could be described as *The cities and the age of the oldest person in the corresponding city.*

In Coldbase group-by is implemented by using an index map. Basically what happens is that the columns involved in the grouping are inserted into a map as keys. The value part of the map are the indexes of the elements (rows) that were inserted. If duplicates are found, these are appended to the values so that each value is actually a vector of indexes. Each value in the map thus corresponds to one group which shares the same grouped values, or key.

Consider the Table 4.6(a). Let's say we would like to count how many persons live in each city. This can be done with the code in Listing 4.4. The group-by is performed by inserting the group-by columns into the map, producing Table 4.6(b). There are three different city names, thus there will be three different groups. The aggregation **Count** simply counts the number of elements in each group, which corresponds to the number

Table 4.5: Example of group by

(a) Table of persons living in different cities.

name	**age**	**city**
John	24	New York
Bill	30	Chicago
Jane	42	New York
Mark	21	Washington
Sarah	16	Chicago
Paul	53	New York

(b) Result when grouping by city and calculating the max age of the persons in each city, from Table 4.5(a)

key (grouped column)	**max(age)**
New York	53
Chicago	30
Washington	21

Listing 4.4: Example query with group-by. Counts the number of persons in each city, from the table in Table 4.6(a) by grouping on **city**.

```
AbstractTable result;
result = new Select(new Column[]{new Column("city"),new Count("name")}, table, null,
                   new Column[]{new Column("city")});
```

of rows that matched a specific key. The **city**-column to the result is taken directly from the key part of the map.

All aggregations can not operate directly on the indexes however, and need to lookup the real values using the row numbers stored in the map. For example if **max(age)** was calculated, the age column would have to be indexed with the rows for each group. First selecting rows $[0, 2, 5]$ and calculating the max of those elements, producing the max value 42, and then selecting the rows $[1, 4]$ and calculating the max for those values, producing 30, etc.

If multiple columns are grouped then each unique combination of the grouped values form one group. For example grouping Table 4.6(a) by both **city** and **age** would produce five different groups.

Table 4.6: Illustration of map when performing group-by.

(a) Table of persons living in different cities.

name	**age**	**city**
John	24	New York
Bill	30	Chicago
Jane	53	New York
Mark	21	Washington
Sarah	16	Chicago
Paul	53	New York

(b) The map created when performing grouping by **city** of Table 4.6(a). The **key**-part are the columns that are being grouped and the **value**-part are the rows numbers (element indexes) that match the corresponding key.

key (grouped column)	**value** (rows)
New York	$[0, 2, 5]$
Chicago	$[1, 4]$
Washington	[3]

4.5 Multi threading

Being able to perform requests in multiple threads, and preferably on multiple cores or CPUs, can potentially increase the potential maximum throughput. Also important is the increased responsiveness for requests. In a strictly serial system queries can be queued and depending on the scheduling a relatively short query might take a very long time to complete only because a time consuming query was started before it. This can severely cripple the experienced responsiveness of the system for the user.

Multi threading solves this by allowing the requests to run seemingly in parallel. True parallelism is achieved in the case where multiple processing units are available and the threads can run on different units. In this case there is also a potential speedup compared to the a serial implementation. A serial implementation can never benefit from more CPUs or more cores since it will only be able to utilize one. A multi threaded design can theoretically be n times faster if the number of processing units are increased by n times, given that the added ones have equal processing power. In reality there are however synchronization issues, overheads and memory bandwidth which often makes the speedup scale less than linearly.

For serial solutions the only way to increase performance is to increase the processing power of the one processing unit that is used. For multi threaded designs the processing power of the units can be increase as well, but also the number of units. The conclusion is that in order to make the system reasonable scalable, with performance in mind, it must first of all be capable of using multiple threads. Secondly, the multi threaded implementation must not introduce overheads so large that the speedup is lost. Finally the concurrency control must be of a granularity such that the overhead is minimized, but still allowing for concurrency when threads use shared resources.

4.5.1 Concurrency in Coldbase

In Coldbase threading supported by using the Java threads API. Synchronization is provided automatically through the table interface and through the query system. Direct manipulation to columns must manually be synchronized if the columns are shared by multiple threads.

A lock can have three different levels, *read*, *append* or *write*. The compatibility between the levels is shown in Table 4.7. Only one append lock can be acquired, however multiple readers can be present during appends. Writes are completely exclusive.

Table 4.7: Lock level compatibility.

	Read	Append	Write
Read	Y	Y	N
Append	Y	N	N
Write	N	N	N

The fact that most data is read and that most writes are actually appends motivates the *append* level to the locks. This concurrency control is relatively coarse grained to keep the overhead involved with locking mechanisms low. There is always a tradeoff between the granularity of the concurrency control and amount of overhead introduced.

The typical workload includes many readers performing different analytical read only queries and typically one input stream per table, adding new data. The lock described above allows for one appending process and many readers simultaneously while keeping

the overhead at a minimum, which fits the expected workload perfectly. Updates are handled by locking the table completely using the write lock but these are considered to be very rare so this is still acceptable.

The locking mechanisms are implemented on top of the Java `ReentrantReadWriteLock` which supports the common read-write lock with shared read locks. To extend it with the append functionality, a complimentary `ReentrantLock` is added. When an append lock is acquired the locking is made in two steps, first the read lock is acquired and then the append lock is acquired.

Locking a read lock guarantees that no writers are present while the append lock makes sure that no other appending processes are present. Releasing an append lock follows the same algorithm but in the reverse order.

If a thread is suspended because a lock could not be acquired it is put in a queue. When the lock becomes available the thread will be resumed. If multiple threads are waiting on a lock, the lock will be allocated in a *first come first served* fashion.

4.5.2 Locking facility

In some situations many locks must be acquired before an operation can take place. It is then important to provide atomicity so that either all locks are aquired, or no locks at all and the thread is suspended. An example when this is necessary is the left outer join operation (see Section 4.3.1) where both tables must be locked for reading. This can be done atomically by using the implemented lock handler.

The interface is simple, just provide the locks you wish to manipulate and the operation (lock or unlock). If all locks can not be acquired the thread is put in a wait queue and is signaled to try again as soon as a lock is released. To make sure threads get these signals *all* unlocking operations must go through this interface.

4.5.3 Appends to keyed tables

In Section 4.3 it was mentioned that appends to keyed tables acts as updates if the key already exists. In this case an append lock is not enough and a write lock is used instead.

4.5.4 Concurrency in tables and queries

When rows are appended to a table, the row count variable is always increased after the append is completed for all columns. This way, it is always safe to read the number of elements that are specified in the table from each column at any time if a read lock is first acquired. The `Select` query, for example, locks the table for reading and gets the number of rows. Only the number of rows read at this point can be addressed during the query. This provides a kind of snapshot-isolation from other threads that might be appending more data to these columns simultaneously.

When the underlying array in a vector must grow they are typically re-allocated twice the old size and the old contents is copied to the new array. Other threads will be using a snapshot of the data and have a reference to the old array object. This is perfectly fine however since the Java garbage collector will only remove the old arrays when they are no longer referenced. This gives a natural snapshot isolation between threads for growing vectors.

Chapter 5

Results

This chapter includes a few short examples of how Coldbase can be used and an evaluation of the implementation.

5.1 Introduction to Coldbase usage

This section uses a few examples to explain how to create a table, insert data and construct queries against the table. A complete user's guide is attached as Appendix A.

5.1.1 Creating tables and inserting data

Creating tables is done by simply creating a table object, specifying the column names and the data types of the columns. Creating a table from already existing columns is also possible. Setting a key constraint for one or more columns is done by calling the `makeKeyColumns` and supplying the names of the columns. An example creating a table with three columns and making the first column keyed is displayed in Listing 5.1.

To insert data into the table made in Listing 5.1 the `append` or `bulkAppend` methods can be used. `append` can be used to insert one row while the `bulkAppend` can insert several rows at once. If an inserted row has a matching key as a row in the table, it will update the corresponding row in the table instead, thus the insert instead implicitly becomes an update operation.

5.1.2 Making queries over the table

The most common query is the `Select`. The constructor takes four arguments: the columns to select, the source (table or output from another query), conditions and groupings.

Listing 5.1: Creating a keyed table

```
Table table = new Table(new String[]{"id", "desc", "value"},
                        new DataType[]{DataType.INT,
                                       DataType.STRING,
                                       DataType.DOUBLE});
table.makeKeyColumns(new String[]{"id"});
```

Listing 5.2: Example of a simple query

```
Select select = new Select(new Column[]{new Column("desc"), new Sqrt("value")},
                           table, null, null);
AbstractTable result = select.execute();
```

If no conditions or groupings are wanted, `null` can be passed. The columns to select can simply be a `Column` object to get the column as it is, or any of its subclasses. For example using a `Sqrt` object over a numeric column will return it square-rooted. This is very similar to SQL syntax. The example in Listing 5.2 shows a simple query that takes the square root of the value column and also selects the description column.

5.1.3 Threading and synchronization

When using the query system in Coldbase, locking and synchronization is already integrated. This means that if two Java threads are created and performs queries, updates, appends, etc. over the same table they will be automatically synchronized where necessary and run in parallel when possible. The interface to vectors is not synchronized however, so getting a vector out of a table and performing operations on it is potentially dangerous. If such an operation is necessary the user is required to use the locking mechanisms explicitly. Examples on how to do this are shown in Appendix A.

5.1.4 Grouping

Grouping basically works like the GROUP BY clause in SQL. One notable difference is that if the result is grouped but no aggregations are made, the returned data will be in form of vectors. Each element is then a vector of elements with the same length as the number of tuples in that group. Generally some kind of aggregation is made on grouped data however. Consider Table 5.1. This table has timestamped transactions to millisecond precision. A typical analytical query could be to count the number of transactions done each minute in the system. An example query is given in Listing 5.3 that accomplishes this using groupings and aggregates. This is done by performing grouping on the time column where the time value has been rounded to even minutes. This will put all transactions for a specific minute in one group. The count aggregate is performed over the groups and will result in the number of transactions for each minute.

Table 5.1: Table with timestamped transactions.

time	id	count
08:20:00.000	2	200
08:20:01.241	1	231
08:20:01.735	2	100
08:20:02.241	10	31
08:20:03.411	4	5
08:20:04.600	30	20
⋮	⋮	⋮

Listing 5.3: Selects the number of transactions performed each minute from Table 5.1 and the timestamp of the first transaction for each minute.

```
Select select = new Select(new Column[]{new First("time"), new Count("id")},
                           table, null, new Columnd[] {
                           new Round(new Column("time"),
                                     new Atom(new TimeVector("00:01:00.000")))});
AbstractTable result = select.execute();
```

5.2 Interface and API evaluation

The composite pattern-based [9] query interface makes query construction intuitive and easy to integrate with Java applications. Compared to SQL it requires a bit more typing to create the objects required to form a query, but with a Java-capable IDE with completion this pain is reduced. A good IDE may even make query construction easier by providing inline documentation based on the javadoc-tags in the source code and smart completion to suggest compatible class types.

The interface for the primitive vector types can also be used directly to perform more obscure operations that may not be implemented at the query level. It is also rich enough to provide all of the functionality that is provided by the query system. This is because the query system itself is only built on top of the primitive vector interface.

Building an SQL parser and providing SQL support on top of the current query system is very simple since the object composition is very similar to how an SQL query is composed.

Since this implementation is mainly a prototype to be used for evaluation, the interface is somewhat sparse in some areas. Some constructor overloading remains to be done in order to make query composition even simpler, and not all functionality is implemented. The design of the interface allows for easy extension and it is very easy to add more functionality by constructing more classes.

5.3 Code maintainability

In Section 3 the downside of using primitive type arrays as storage was discussed. The downside is that generics can not be used and type parameterization and generalization of classes is not possible. Another consequence is that method dispatching is not possible either. As a result, one class for each primitive type had to be implemented. The interface to the primitive types is quite rich in order to benefit from the low-level access to the arrays and get good performance. This, however, results in a lot of re-implementation of different methods for all the different types. For example, multiplication is not very different for an integer type and a floating point type, yet it must still be implemented for both of them separately.

All the type-specific methods add code to the primitive types and cripples the maintainability of the code. For example if a primitive method `foo` was to be added to the numeric type interface and it requires access to the underlying array, it would have to be implemented by all numeric vectors (currently only integer and double). Also if the behavior of some method is to be changed, it may have to be changed for all types.

This is the result of using primitive types, but as the Java evaluation showed in Section 3, it is necessary to use the primitive times to limit the memory overhead.

Once the primitive type vectors are stable and do not require further changes or

additional functionality this is not an issue however. At the query system interface all types are abstracted into abstract vectors and the underlying arrays are of no concern.

5.4 Query performance

Performance has been measured by timing a few queries. The queries were supplied by Nomura and reflect some typical query workloads in their database systems. The same queries were tested in PostgreSQL and in the Q programming language[1].

The queries were performed on the quote data set (see Appendix B.1) which contains about 1.2 million rows of data. The benchmark details and results are attached in Appendix B.4.

For this type of query processing it is clear that a conventional row-oriented database is no match for a column-oriented one. The Java implementation introduced here is outperformed by the Q array processing language by a factor of up to four times. For the left join benchmark this implementation almost performed as well however, which is much thanks to the key map implementation described earlier. Q is considered to be an extremely efficient and fast processing language so it is not surprising that it performs well. It has been refined during years of development and was written in C. This implementation still performs very well considering that it is written in Java and has only been under development for a few months. In fact, getting the performance showed in the benchmarks is in many ways better than what was expected. This implementation also supports multi threading as opposed to Q which is sequential. More on the multi threading evaluation in Section 5.5.

Compared to the timings by PostgreSQL this implementation is much faster and proves that the column oriented structure is much more efficient for the type of queries tested.

5.4.1 Finding the bottlenecks

The Java performance analysis showed that performing operations on the primitive arrays is about as fast in Java as in C. Where are the bottlenecks then, one might ask. The code has been profiled and analyzed to determine where the biggest bottlenecks are. The primitive operations are very fast, as showed in the Java performance comparison in Section 3, so this is not the problem. Profiling has proved that in most cases the bottleneck lies in excessive copying of data. The idea behind the interface of the primitive types is that operations can be consecutively typed in the form `vector.operation1().operation2()`. Each operation typically returns a newly allocated vector for the result. Now consider the following example:

```
column.mul(vec1).add(vec2);
```

The statement will allocate one intermediate vector for the result of the multiplication with `vec1` and then another copy for the result when adding `vec2`. The intermediate vector is allocated and then simply thrown away. The same problem of intermediate realizations of data is present when the nested objects are used with the query system. Possible solutions to this are discussed in Section 5.4.2.

Also indexed elements are always realized. Consider a select query matching most elements of a large table. Before any operations take place on a column, all the selected

[1] Q is built on top of the K language which is used in the KDB+ database[2]. It provides a syntax that is closer to natural language than the K programming language does.

elements are copied into a new vector only to be thrown away as soon as the operation is done with it. The reason it was designed this way was to reduce complexity. Also if a column is used by multiple objects, the copies can not be shared. One copy of the addressed column is made for each object.

5.4.2 Possible solutions

A solution to the problem is to make the intermediate operations in-place so that only the result vector is allocated once. In theory only the first operation should need to allocate a result, and then all following operations could modify this vector by performing their operations in-place. In the query system the first operation is typically an index operation that selects the elements that passed the conditions.

There are a few complications with realizing such an implementation however. First of all type the coercions might require extra realizations. For example, consider the following operation where both `vec1` and `vec2` are integer vectors:

```
vec1.add(vec2).sqrt();
```

First a new integer vector is created for the result of the `add` operation. The square root operation returns a double vector however, so the previous integer vector can not be used as return type so the square root operation must allocate a new vector.

Another problem is how it should be integrated into the interface. How are the called operations to know if it should be executed in-place or if a result should be allocated. It could be an attribute in the vector telling if the instance is a copy or not. In the first example, the `mul`-operation would see that `column` is not a copy and allocate a new vector for the result. The newly allocated vector will have the copy attribute set. The `add` operation on the new vector will notice that it is a copy and it can be executed in place returning the same vector. The problem with this approach is that the attribute must somehow be cleared after the last operation. Requiring the user to do so explicitly would surely cripple the otherwise clean interface. It could be the other way around however, requiring the user to set the attribute whenever operations should be made in place. This could speed up occasions where performance is critical and still not make the interface hard to use for the common case. The query system could make use of such an attribute automatically and automatically remove it before returning the result.

Another approach is that the interface is extended a bit so that the result vector can be supplied to the operations. The old methods that allocate the result could remain and simply internally call the new methods with an allocated result vector. The interface for the primitive types would almost contain twice the amount of methods that it contains now however and this would require quite a bit of refactoring of current code.

Focusing on indexing and sharing columns could be made instead. In this case the solution might be to integrate indexing into the interface. Operations would then be formed only over the given elements. For example `vec1.add(vec2, elements)` where `elements` is a vector with indexes. In this case the operation would have to return a copy however, since it is the original column that is being used for the operation. The result is that only the indexing step can be removed. For most query however the nesting depth of operations is seldom more than one, so the indexing step makes up a significant part of the execution time. It could also be combined with the approached described above.

Sharing input is another aspect of minimizing data copying but also the number of operations. Consider the following query:

```
SELECT (col1 + sqrt(col2)), sqrt(col2), col3 * sqrt(col2) FROM table
```

The query flow is illustrated as a tree in Figure 5.1. Computing the square root three times seems like a waste however. In theory it should only have to be computed once and then the calculation can be shared by the other operations. In the case where the input is shared the operations can not be done in place however. The tree using shared data is shown in Figure 5.2. Identifying sharable nodes could be done through a map.

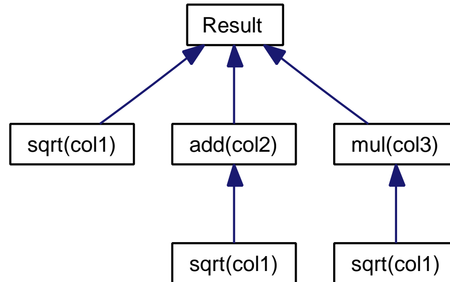


Figure 5.1: Query tree where data is not shared.

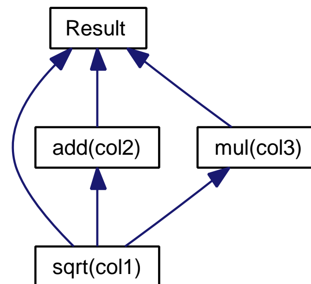


Figure 5.2: Query tree where data is shared.

5.5 Multi threading

The benefits and importance of multi threading are discussed in Section 4.5, pointing out increased responsiveness, speedup and scalability.

The benchmarks performed in Appendix B.5 compares serial and threaded performance. The results show that for operations that are computationally more expensive the gains when using multiple threads are larger, which again points at memory bandwidth as being the limiting factor in some cases.

The benchmarks were run on the Intel dual core and the Sun UltraSparc systems presented in Appendix B. In the dual core system the memory bandwidth is shared by the cores. Thus, if the application performance is limited by the memory bandwidth, using a dual core system instead of a single core may not result in a dramatic improvement to performance. Since the major bottleneck was the copying of data throughout the interface it is very likely that this will also be the bottle neck for multi threaded performance.

The performance for the UltraSparc system shows a much better speedup however. The reason is that the memory bandwidth per processing unit is higher than for the dual core system. Benchmarks were also run on a modern AMD Opteron SMP system with two CPUs with similar speedup to the UltraSparc.

To further investigate the multi threaded performance on the dual core system a small benchmark was set up. The benchmark used two versions of an element-wise vector division method. The code for the two versions are displayed in Listing 5.4.

Listing 5.4: Element-wise division methods for vectors. The first allocates a result vector, the second writes the result to a given result vector.

```
public static double[] div(double[] vec1, double[] vec2) {
  double[] res = new double[SIZE];

  for(int i = 0; i < SIZE; ++i) {
    res[i] = vec1[i] / vec2[i];
  }
  return res;
}

public static void div2(double[] res, double[] vec1, double[] vec2) {
  for(int i = 0; i < SIZE; ++i) {
    res[i] = vec1[i] / vec2[i];
  }
}
```

The first version allocates a vector for the result and writes the result of each division to this vector. The second version instead writes the result to a given result vector. The benchmark measures the execution time for executing two divisions serially over 1M elements and compares it to executing them in parallel. The results are displayed in Figure 5.3 and in Table 5.2. For the `div2` method the result vector is allocated before the start of the timer.

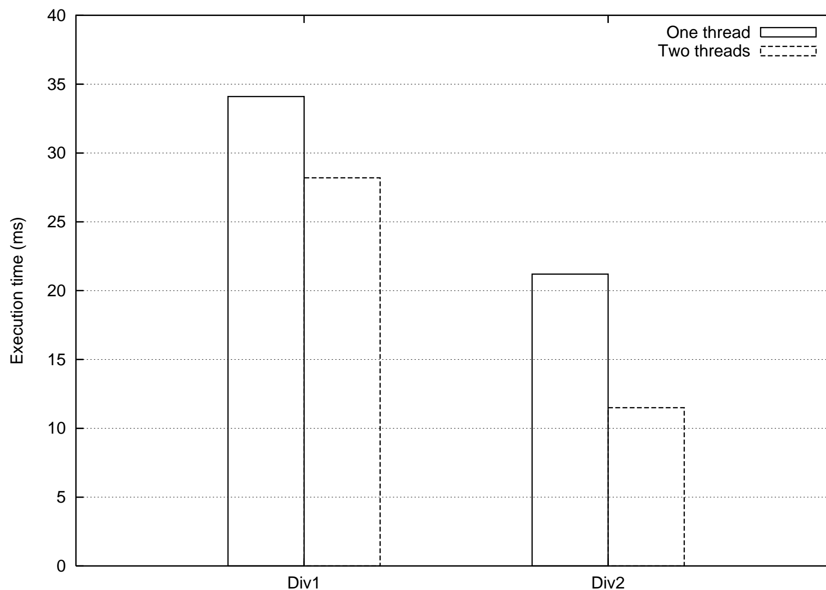


Figure 5.3: Benchmark of the two element-wise vector division methods displayed in Listing 5.4, run on a Intel dual core system.

Table 5.2: Benchmark of the two element-wise vector division methods displayed in Listing 5.4, , run on a Intel dual core system.

	One thread (ms)	Two threads (ms)
`div1`	34.1	28.2
`div2`	21.2	11.5

The speedup can be calculated as $S_p = \frac{T_1}{T_p}$ where T_1 is the execution time of the sequential algorithm and T_p is the execution time of the parallel algorithm with p processors. For this benchmark the speedup when using two threads for the first method is $S_2 = \frac{34.1}{28.2} = 1.21$ and the speedup of the second method is $S_2 = \frac{21.2}{11.5} = 1.84$. It is clear that the memory allocation of the result vector is a limiting factor for the achievable speedup. This further supports the argument that the overall performance on the dual core system is memory bandwidth limited.

If the speedup is calculated for the multi threaded benchmarks over the databases it is between 0.95 and 1.23 for the dual core system, which is not that good. For the UltraSparc system and the AMD Opteron SMP system the speedups are between 1.33 and 1.84, where the 1.33 was for the memory intensive first aggregate query. Both the second aggregation and the primitive vector operations achieved speedups of over 1.8 which can be considered to be very good.

5.5.1 Possible improvements

For the dual core system the limiting factor for the multi threaded speedup is essentially the same as the bottle necks for sequential performance, the same solutions also apply. By restructuring the interface to reduce the average number memory allocations for a query the performance will also scale better using multiple threads.

For the SMP system and the UltraSparc machine with four CPUs the memory bandwidth is higher relative to the processing power per processing unit and the penalty to the speedup is also substantially lower. For this kind of systems a restructuring might not improve the speedup as much, but since the sequential performance is likely to improve the overall performance using multiple threads will also improve.

Chapter 6

Conclusions

This implementation shows how a Java implementation can achieve good performance while also keeping memory usage low. It is optimized for the given workload and provides multi threading for increased responsiveness and scalability. It also benefits from the integration capabilities with existing Java applications.

The query interface is based on object oriented principles and is easy to extend with new functionality. The composition of query objects is similar to that of SQL and should be familiar and easy to learn for users who have used SQL previously.

Performance is good but can be improved. The sequential performance should improve if data was only copied when it is absolutely necessary. In the multi threaded case the speedup should also be improved for system where the memory bandwidth is the limiting factor.

The work on this project has progressed quite nicely. Of course there have been some occasional realizations that has required redesigning and rethinking parts of the design but that is only a natural part of the process. The background study of existing column oriented databases and of common techniques used in the area provided a good starting point, even though only a few aspects were actually implemented.

The field of column-oriented databases has been around for quite some time but has recently attracted more attention, much thanks to the C-store[15] by Michael Stonebraker *et al.* introduced at VLDB[7] in 2005. But this is not the only reason. The fact that many areas have increasing amounts of data and predictable workloads has resulted in that general purpose applications are losing ground, favoring special purpose solutions instead. For these reasons this thesis project has been an interesting and relevant topic to work with. Hopefully it will also contribute to the area or be a foundation for future developments.

6.1 Limitations

The goals that were initially set up for the project have been met. No further restrictions or limitations other than those stated had to be made. Of course improvements and additions can always be made but the initial goal to provide enough functionality to be able to test the implementation thoroughly has been met.

6.2 Future work

There are still many features that can be found in most commercial and commonly used database systems that this implementation lacks. These include transactions, recovery, ability to use disk storage and others. Each one of these features could probably fill the scope of a thesis work by itself.

There is also work to be done to improve currently implemented features. One apparent future improvement is to address the performance bottlenecks that were discussed in Section 5.4.1 and Section 5.4.2. This would make multi threaded performance better for systems with limited memory bandwidth and thereby improve the scalability of the application. Also extending the interface and providing more of the basic functionality is necessary. The implementation is designed to extend easily for such improvements however, so providing more functionality should not require that much effort.

Chapter 7

Acknowledgements

First of all I would like to thank my supervisor at Nomura (Lehman Brothers when this thesis started), Martin Berglund, for his support and advice throughout the entire process. The original thesis proposition was his and his insight in the area has been invaluable, especially during the occasional setbacks in the work.

I would also like to thank my supervisor at the Department of Computing Science at Umeå University, Michael Minock, for his constructive criticism and guidance in both the writing process and during the development.

Last but not least I would like to thank Mårten Forsmark who introduced me to the company and informed me about the possibilities for conducting thesis work at Lehman Brothers.

References

[1] BLAS, Basic Linear Algebra Subprograms. Netlib implementation. `http://www.netlib.org/blas` [Accessed 2009-02-02].

[2] Kx Systems KDB database. `http://www.kx.com/products/database.php` [Accessed 2009-02-02].

[3] LAPACK, Linear Algebra PACKage. `http://www.netlib.org/lapack` [Accessed 2009-02-02].

[4] Daniel Abadi and Samuel Madden. Debunking Another Myth: Column-Stores vs. Vertical Partitioning, 2008. `http://www.databasecolumn.com/2008/07/debunking-another-myth-columns.html` [Accessed 2009-02-02].

[5] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682, New York, NY, USA, 2006. ACM.

[6] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 967–980, New York, NY, USA, 2008. ACM.

[7] Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors. *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*. ACM, 2005.

[8] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems (5th Edition)*. Addison Wesley, March 2006.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[10] Sun Microsystems Inc. The Java HotSpot Virtual Machine, v1.4.1. `http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/Java_HSpot_WP_v1.4.1_1002_1.html` [Accessed 2009-02-02].

[11] Sun Microsystems Inc. Introduction to Autoboxing and introduction to Tables with JDesktop Network Components (JDNC) Tech Tips, April 2005. `http://java.sun.com/developer/JDCTechTips/2005/tt0405.html` [Accessed 2009-02-02].

[12] Intel®. Streaming SIMD Extensions 4 (SSE4) Instruction Set Innovation. `http://www.intel.com/technology/architecture-silicon/sse4-instructions/index.htm` [Accessed 2009-02-02].

[13] Roger MacNicol and Blaine French. Sybase IQ Multiplex - Designed For Analytics. In *VLDB*, pages 1227–1230, 2004.

[14] Vladimir Roubtsov. Java tip 130: Do you know your data size?, August 2002. `http://www.javaworld.com/javaworld/javatips/jw-javatip130.html` [Accessed 2009-02-02].

[15] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-Store: A Column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

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

[17] M. Zukowski, S. Heman, N. Nes, and P. A. Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, Atlanta, GA, USA, April 2006.

Appendix A

User's Guide

This appendix describes how to use the interface to construct queries. It also describes what to expect when it comes to locking, performance, etc. when performing queries, and also what the user is expected to do.

A.1 Vectors

Vectors are used to represent vectors of the implemented types. There exists vectors for integers, doubles, time, date, bit, char, string, vectors and generic (`Object`) types. All vectors implement the interface declared by the `AbstractVector`-class.

A.1.1 Common interface

The following methods are available for all vector types.

- `array` - Returns an `Object`-reference to the underlying array object.
- `append` - Appends a single value to the vector. If the value is of incompatible type an exception will be thrown.
- `bulkAppend` - Appends a vector of values.
- `bulkUpdate` - Updates the values of the given indexes with the supplied values.
- `clone` - Creates a deep copy of the vector.
- `coerceList` - Converts an atom-type vector into a list type of the same length as the vector argument. This method is used by the internal automatic coercions.
- `find` - Finds the first occurrence of the sought value. The index of the element is returned, or -1 if it was not found. The method uses linear search by default but can also use binary search for sorted vectors.
- `find` (vector) - Finds the first occurrence of each sought value. Returns a vector of indexes of the sought values or -1 for the values that were not found.
- `getLength` - Returns the number of rows in the table.
- `getDataType` - Returns the data type of the vector as the enumerated type `DataType`.

- `hash` - Returns an integer vector with the hashes of the values in the vector.
- `index` - Returns the value at the given index. If the index is out of bounds an exception will be thrown.
- `index` (vector) - Returns a vector of the values at the supplied indexes. If an index is not found an exception will be thrown.
- `indexNulls` - Returns a vector of the values at the supplied indexes. If an index is not found, the null value of the data type is return instead. For a table of null-representations, see Table 4.2.
- `isAtom` - Returns true if the vector is an atom.
- `makeAtom` - If the vector contains only one element it can be converted into an atom type by calling this method.
- `newInstance` - Returns a new instance of the vector. Since each subclass implements this method a copy of the same type can always be obtained even if the type is not known. As parameter the initial capacity of the new vector can be specified.

A.1.2 Numeric vectors

Numeric vectors are a subclass that implement methods for numeric operations. Both `DoubleVector` and `IntVector` implement the interface which includes the following methods:

- `add` - Returns a vector with the values in this vector element-wise added with the given vector.
- `sub` - Returns a vector with the values in this vector element-wise subtracted with the given vector.
- `div` - Returns a vector with the values in this vector element-wise divided with the given vector.
- `mul` - Returns a vector with the values in this vector element-wise multiplied with the given vector.
- `sqrt` - Returns a vector with all values in this vector square rooted.
- `round` - Returns a vector with the values in the vector rounded down to a multiple of the given value.
- `sum` - Returns the sum of all the elements in the vector.
- `coerceTypes` - Takes two numeric vectors as parameters and returns the resulting data type if a numeric operation were performed between the two. For example an integer type and a double type would return a double type.
- `coerceInt` - Coerces the vector into an integer vector.
- `coercedouble` - Coerces the vector into a double vector.

If a numeric operations is made between vectors of different types, automatic type coercions take place. For example multiplying an integer vector with a double vector will result in a double vector. Also unary operations may return coerced data, such as performing a square root operation on an integer vector will result in a double vector.

The numeric vector interface also implements the `Comparable` interface which adds the following methods:

– `geq` - *Greater than or equal*-comparison. Returns a bit vector where the set bits represent the elements in the vector that were greater than or equal to the corresponding value in the argument vector.

– `gt` - *Greater than*-comparison.

– `leq` - *Less than or equal*-comparison.

– `lt` - *Less than*-comparison.

– `neq` - *Not equal*-comparison.

– `eq` - *Equal*-comparison.

– `intersect` - Returns the intersection between this vector and the given vector. The return type is a bit vector where set bits represent values in this vector that the other vector contained. This is a $O(n \log n)$ operation for non-sorted vectors and a linear operation if both vectors are sorted.

– `min` - Returns the index of the minimum value element.

– `max` - Returns the index of the maximum value element.

A.1.3 Time vectors

Since time vectors are derived from the `IntVector`-class it also inherits all the numeric operations. This allows for adding to times, subtracting from times, etc. as if it were integers. The time is representing in milliseconds. The constructor of the time vector can create times from integers containing millisecond time values, or from a string. The string should have the format "HH:MM:SS.sss". If the string could not be parsed a `ParseException` is thrown. Appending values to the time vector is done through the integer vector interface so appended times must expressed as milliseconds from 00:00:00.000.

A.1.4 Date vectors

The date vector is derived from the `IntVector`-class since it stores the date as an integer representing the number of days since 1970-01-01. Since it inherits the integer vector interface dates can be added, subtracted, compared, etc. the same way as integers.

A.1.5 String vector

Strings are internally stored as integers representing indexes into a lookup table, the string internalization table. The single value append method allows for Java `String` insertion and will automatically lookup the corresponding integer in the lookup table. If it did not contain the string it will automatically be added. The bulked append

method does not yet implement this type of behavior and requires that the inserted data is integers.

Comparing string vectors is possible since it implements the `Comparable` interface. The `eq` and `neq` methods are particularly fast since they can be done over the integers, not needing to lookup the actual strings in the table. Other comparisons require lookups and then string comparisons however.

A.1.6 Bit vector

The bit vector implements the `Comparable` interface which gives it the comparison methods. Many of the methods are not very useful on a bit vector however. The methods of interest are the following:

- `where` - Interprets the bit vector as a bit index and translates it into a index vector. For example the bit vector $[0, 0, 1, 1, 0, 1]$ would result in $[2, 3, 5]$, i.e. the indexes of the set bits.
- `and` - Bit-wise *and* operations taking another bit vector as argument.
- `or` - Bit-wise *or* operation.
- `not` - Bit-wise *not* operation.

A.1.7 Char vector

The character vector type stores the Java primitive type `char`. A character vector can be inserted into a string vector as one element, thus it is interpreted as a string.

A.1.8 Vector of vectors and generic vectors

The `VectorVector` and `GenericVector` stores an array of references. The generic version is capable of storing any object type while the vector version stores `AbstractVector` objects. Both classes provide the basic vector interface.

A.1.9 Additions to integer vectors

Since the integer vector is used internally for representing indexes it has a few additional methods.

- `createElementVector` - Creates a vector counting up from 0 to n where n is the parameter to the function. This method is useful when it is necessary to creating index vectors containing all elements. Also a version that uses an offset is implement. This method creates a vector with values from s to $s + n$.
- `modulo` - Modulo operation on the vector.

A.1.10 Examples

Listing A.1: Example of automatic coercion of numeric types.

```
IntVector v1 = new IntVector(new int[]{1,3,5,7}, true);
DoubleVector v2 = new DoubleVector(new double[]{1.1, 2.2, 3.3, 4.4},true);
DoubleVector result = v1.mul(v2); //Automatic coercion to double
```

Listing A.2: More extensive example for manipulating dates.

```
//Date vector with 2008-11-20 to 2008-11-23
DateVector dv = new DateVector(new int[]{14203, 14204, 14205, 14205})

//Add one year (365 days)
dv = dv.add(new IntVector(365));

//Now check if any of the days is a Saturday, since day 0 is a
//thursday the following day numbers apply:
// 0 : thursday,
// 1 : friday,
// 2 : saturday, etc.

//By doing modulo 7 we get the day of the week
BitVector isSaturday = dv.mod(7).eq(new IntVector(2));

//Get the indexes of the saturdays, if any
dv.index(isSaturday.where());
```

A.2 Tables

Tables map column names to the actual vector storing it. It has the interface for inserting rows, i.e. inserting values to each column. It also provides locking and the possibility to key columns. The interface includes the following methods:

- `append` - Appends one row to the table. This method will implicitly lock the table for appends.
- `bulkAppend` - Appends many rows to the table. The argument is an array of vectors. The first vector will be appended to the first column and so on. This method will implicitly lock the table for appends.
- `getColumns` - Returns the underlying vectors. This should generally not be used since it can violate the locking. The user must provide external locking manually.
- `getColumnTypes` - Returns the data types of the columns.
- `getColumnNames` - Returns the names of the columns. An integer array can also be passed to only get the names of specific columns.
- `getLock` - Returns the lock for this table, for external synchronization.
- `getRow` - Returns a single row from the table. This method should not be used when fetching several rows or iterating the table since it is not very fast. This method implicitly locks the table for reading.
- `getRowCount` - Returns the number of rows in the table.
- `loj` - Left outer join on another keyed table. The keyed columns in the operand must exist in this table since the join is implicitly performed over these columns. This method locks both tables for reading using the locking facility.
- `makeKeyColumns` - Makes the specified columns keys for this table.

The lock used for the table is a three level lock for reading, appending or writing. For lock compatibility, see Table 4.7. The `getLock` method can be used to fetch the lock in case the table must be externally synchronized. This can be the case when

the columns must be directly manipulated, and not through the table interface. It is then important to lock the table owning the columns so that other threads can not be performing conflicting operations simultaneously. The user must also make sure to use the locking facility when unlocking locks. For more information about manual locking and the locking facility, see Appendix A.3

A.2.1 Keyed tables

When performing appends into a keyed table the table is automatically locked for writing. This is necessary because appends might actually be updates in case the key part of the appended row already exists in the table. In this case the new row replaces the old one in the table.

Keys can be made over one or more columns. Making a non-empty table keyed will discard any duplicate key rows.

A.2.2 Examples

Listing A.3: Example of how to create a simple keyed table.

```
//Create a table with id, name and price columns
Table table = new Table(new String[]{"id", "name", "price"},
    new DataType[]{Datatype.INT, DataType.STRING, DataType.DOUBLE});

//Make the id-column key
table.makeKeyColumns(new String[]{"id"});

//Insert a single row
table.append(new Object[]{
    new Integer(0),
    new String("Foo"),
    new Double(10.2)});
```

Listing A.4: Example of how to create a simple table from preexisting data.

```
//Create two vectors
IntVector vec1 = new IntVector(new int[]{1, 4, 2, 6}, false);
DoubleVector vec2 = new DoubleVector(new double[]{2.5, 1.26, 2051.5, 35.0});

//Create a table of the columns
Table table = new Table(new String[]{"num", "value"},
    new AbstractVector[]{vec1, vec2});
```

A.3 Manual synchronization

Manual synchronization is required when manipulating columns directly since the vector interface does not provide automatic synchronization. If the vectors are part of a table the corresponding lock should be used for locking. This guarantees that other threads interacting with the table will not be in conflict with the operations performed directly onto the columns.

When locking more than one lock the lock handler must be used. This class has static methods for atomically locking one or more locks in a all-or-nothing fashion. If not all locks can be acquired at once the thread is suspended until a lock becomes available. In order for this to work properly it is important that the facility is used when unlocking locks. This will signal the waiting threads to try again.

The three level lock `ReadAppendWriteLock` extends the Java `ReadWriteLock` with the addition level for appending. It has the following interface:

- `appendLock` - Locks the append lock.
- `readLock` - Locks the read lock.
- `tryAppendLock` - Tries locking the append lock.
- `tryReadLock` - Tries locking the read lock, if it was already lock, *false* is return.
- `tryWriteLock` - Tries locking the write lock.
- `writeLock` - Locks the write lock.

Since unlocking should always be made trough the lock handler these methods are protected. The lock handler has the following interface:

- `lock` - Tries locking an array of locks at the specified levels. The calling thread is suspended if all locks could not be acquired.
- `unlock` - Unlocks the single or array of locks specified with the specified levels.

The lock handler methods are synchronized to be atomic.

A.3.1 Examples

Listing A.5: Example of how to externally synchronize vectors.

```
void someMethod(Table t) {
  /*
   Lets say we know that the underlying vector is an
   integer vector and we wish to manipulate the data
   directly
  */
  IntVector col1 = (IntVector) t.getColumns()[0];
  int[] array = (int[])col1.array();

  //Lock the table
  ReadAppendWriteLock lock = t.getLock();
  lock.writeLock();
  try {
    //Manipulate array
  } finally {
    //Release lock and signal threads
    LockHandler.unlock(new ReadAppendWriteLock[]{lock},
        LockHandler.LockTypes.WRITE);
  }
}
```

Listing A.6: Example of locking and unlocking multiple locks.

```
ReadAppendWriteLock lock1, lock2;
lock1 = someTable.getLock();
lock2 = otherTable.getLock();

//Lock both locks for reading

LockHandler.lock(new ReadWriteAppendLock[]{lock1,lock2},
    new LockHandler.LockTypes[] {
        LockHandler.LockTypes.READ,
        LockHandler.LockTypes.READ}
try {
```

```
    //Access data
} finally {
    //Release lock and signal threads
    LockHandler.unlock(new ReadAppendWriteLock[] {lock,
        lookup.getLock()},
        new LockHandler.LockTypes[] {
            LockHandler.LockTypes.READ,
            LockHandler.LockTypes.READ});
}
```

A.4 Query interface

Constructing queries is a simply matter of object composition. To make a SELECT query the `Select`-class should be instantiated. The source can either be a table, another `Select`-object, or any other object derived from `AbstractQuery<AbstractTable>`). The constructor takes four arguments, the operations, the input, the conditions and the groupings.

A.4.1 Operations

The operations that can be used are the following:

- `Column` - Simply returns the column as it is.
- `Arithmetic` - Numeric operations such as multiplication, addition, etc. Operates over two columns.
- `Aggregate` - Base class for aggregate operations
  - `Count` - Returns the number of elements in the source, or for each group if groupings are present.
  - `First` - Returns the value of the first element in the source or in the group.
  - `Last` - Returns the value of the last element in the column or group.
  - `Max` - Returns the maximum value of the column or group. Only applicable for types implementing the `Comparable`-interface.
  - `Min` - Returns the minimum value of the column or group. Only applicable for types implementing the `Comparable`-interface.
  - `Sum` - Returns the sum of all the elements in the column or for each grouping.
- `Functions`
  - `Round` - Rounds the column down to a multiple of the given value.
  - `Sqrt` - Returns the square root of the column or groups.

A.4.2 Conditions

Conditions return a bit vector. They are typically used as the WHERE-part of the query but can also be used as an operation. In this case the resulting column is a bit vector column. The common base class for conditions is the `Condition`-class.

- `Comparison` - Comparisons such as less than, equal to, etc.

– `In` - Returns true for elements in the column that are contained within the specified vector.

– `Within` - Range condition. Returns true for elements that are within the specified range.

– `Not` - Bit-wise *not* of a bit vector column.

A.4.3 Grouping

Grouping is done by simply using operations with the group parameter to the select. The result of the grouping operations will define the key of the result.

A.4.4 Examples

Listing A.7: Example query with a simple condition.

```
//SELECT id, sqrt(price) FROM table WHERE price < 5;
Select sel = new Select(new Column[]{new Column("id"), new
  Sqrt("price")}, table,
    new Condition[]{
      new Comparison("price".
                     Comparison.Type.LT,
                     new Atom(new IntVector(5)))
    }, null);

//Execute the query
AbstractTable result = sel.execute();
```

Listing A.8: Example query with groupings.

```
//Count how many employees live in each city
Select sel = new Select(
    new Column[]{new First("city"), new Count("city")},
    table,
    null,
    new Column[]{new Column("city")}); // group by city
```

Listing A.9: Example query rounding times.

```
//Selects the products from table with time column rounded to even minutes
Select sel = new Select(new Column[]{
    new Column("product"),
    new Round("time", new Atom(new TimeVector("00:01:00.000"))),
    new Column("price")},
    table,null,null);
```

Listing A.10: Example query with nested operations.

```
//SELECT price * 2 FROM table WHERE not (price < 22.5
// and price > 10.0);
Select sel = new Select(new Column[]{
    new Sqrt(
        new Arithmetic("price",
                   Arithmetic.MUL,
                   new Atom(new IntVector(2))))},
    table,
    new Condition[]{
        new Not(new Within(
        new Column("price"),
        new Atom(new DoubleVector(10.0)),
        new Atom(new DoubleVector(22.5))
        ))
    }, null);
```

A.5 Utilities

A.5.1 Printing tables

The `FormattedPrint` utility class was implemented for formatted printing of tables. It translates the null values into the "null" string in the output, looks up internalized strings and formats date and time vectors into string representations.

It should be considered as a testing and debugging utility however and is not a good solution for dumping large tables into text files for example.

Listing A.11: Example of printing tables.

```
AbstractTable table = query.execute();
// Prints contents to standard out
FormattedPrint.printTable(table);
```

A.5.2 Timing queries

Timing queries can be done manually using any of the Java timing methods, or by using the implemented `Timer`-class. The class implements a basic stopwatch interface which can start, pause, reset and resume the timer. The resolution is milliseconds, unless the platform or Java implementation has other restrictions. It is based on the `System.currentTimeMillis()` time.

Listing A.12: Example of query timing.

```
Timer timer = new Timer();
timer.reset();
query.execute();
long time = timer.getTime();
```

Appendix B

Benchmarks and results

This appendix chapter covers various benchmarks performed during the design and evaluation phases of the project.

The benchmarks were run on an Intel Core 2 Duo 3.0GHz running Ubuntu Linux v8.04 with 2GB RAM. The multi theading benchmarks were also run on a Sun 4x UltraSparc IIIi, 1593MHz machine with 16GB RAM running SunOs 5.9.

B.1 Quote data

Nomura has supplied quote data which represents a typical domain. The data shows the bid and ask prices for different symbols at different times. The `asize` and `bsize` columns shows how many bids and asks are present at the corresponding price. Table B.1 shows the first 20 rows from this data file. The data has been changed from its original form so that it does not represent the real world data. It is still represents typical quote data however. The data file contains a total of 1 196 698 records.

B.2 Primitive type loops in Java and C++

This benchmark was discussed in Section 3.2 and compares the performance of a simple application performing basic operations over an array of primitive type values. The benchmarks were run with Java HotSpot VM version 1.6.0_10 and the C++ code was compiled with GNU g++ 4.3.2. The following compiler flags were used: `-O3`, `-funroll-loops`, `-floop-optimize`, `-mtune=core2 -msse`, `-msse2`, `-mss3`, `-mmx`, `-mfmath=sse`

It should be noted that the result varied with a few milliseconds between different runs. Sometimes the Java version was a bit faster, and sometimes the C++ version was faster. The execution times are averaged from ten runs and displayed in Table B.2.

B.3 Hash map benchmark

This benchmark compares the implemented vector based hash map to an implementation based on the Java `HashMap`. This benchmark represent typical cases for the database and

Table B.1: Excerpt from quote data.

date	time	sym	bid	ask	asize	bsize
2008-07-01	08:00:25.666	XMY	5870.57	5871.01	2	13
2008-07-01	08:00:25.666	IKA	6.45	8.44	55	110
2008-07-01	08:00:25.708	FHX	185.59	187.37	8388	13858
2008-07-01	08:00:25.708	EKW	5870.57	5871.01	2	13
2008-07-01	08:00:25.709	XCO	231.55	233.54	1375	7696
2008-07-01	08:00:25.709	VSI	30.54	30.55	1155	69
2008-07-01	08:00:25.737	LHY	7205.68	7207.9	8	21
2008-07-01	08:00:25.737	EWN	98.64	98.65	1286	946
2008-07-01	08:00:25.737	RJS	98.29	98.31	1286	203
2008-07-01	08:00:25.758	YDK	152.12	152.25	383	1340
2008-07-01	08:00:25.758	VBO	10407.36	10469.52	0	0
2008-07-01	08:00:25.758	HAU	20.34	20.42	2200	27500
2008-07-01	08:00:25.787	BOB	2535.24	2548.56	1210	11002
2008-07-01	08:00:25.787	ISW	152.11	152.43	550	22
2008-07-01	08:00:25.787	HYV	7128.42	7129.75	8	21
2008-07-01	08:00:25.787	XDC	2152.51	2165.83	6602	6602
2008-07-01	08:00:25.787	DKT	1975.8	1997.11	6602	6602
2008-07-01	08:00:25.787	CVI	7128.42	7129.75	8	21

Table B.2: Benchmark results for operations on primitive type arrays.

Operation	Java time (ms)	C++ time (ms)
Sub	29.1	27.8
Sub unrolled	30.0	27.5
Div	66.1	62.8
Div unrolled	66.7	62.5
Comp	18.8	20.0
Comp unrolled	19.6	19.7

does not necessarily prove that the implemented hash map is better for all cases, such as row-by-row operation. The benchmarks were run with Java HotSpot VM 1.6.0_10

B.3.1 Key map

The key map maps a set of values to a single integer value, typically representing the row number. If an insert is made with a key that already exists in the map, the data will be replaced. The key map is primarily used for keyed tables and for the left join operation.

This benchmark uses the quote data (see Section B.1) to measure the performance of an insert into the key map. A corresponding operation is also made using a Java `HashMap`. The time is measured when inserting a number of rows into a map already containing several rows.

The first benchmark inserts non-overlapping data into an empty key map so that for n inserted records, n new mappings are created. Both the implemented and the Java map have initial capacity large enough to avoid rehashing during the insert. The results

are displayed in Figure B.1. The results show that the implemented `KeyMap` clearly outperforms a corresponding implementation based on the Java `HashMap`.

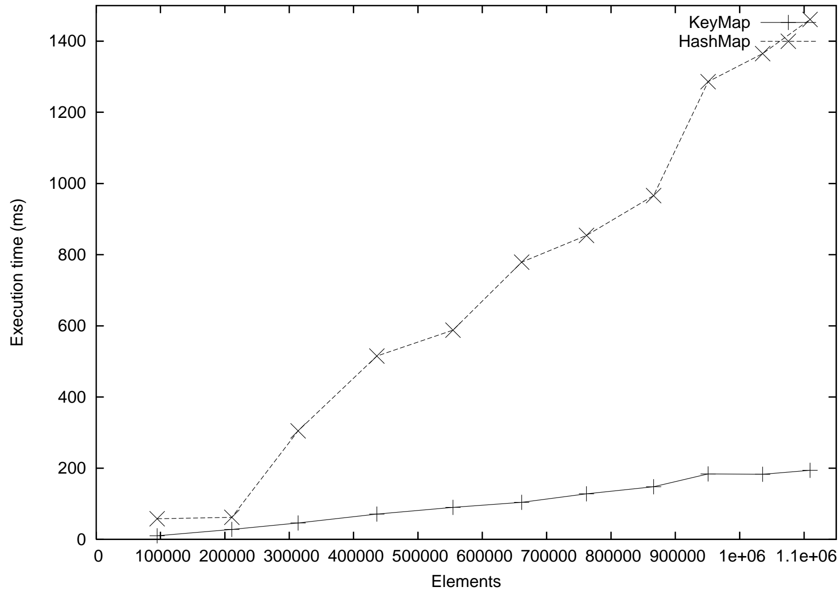


Figure B.1: Benchmark of the implemented `KeyMap` vs. Java `HashMap` with non-overlapping data.

The second benchmark inserts records into a map already containing 280560 records. The initial table contains all distinct (**sym**, **time**) pairs from the quote data where the time has been rounded to 10-second intervals, hence the odd number of records. Selecting and inserting all (**sym**, **time**)-pairs with time rounded to 5-second intervals will result in about 50% updates and 50% inserts. This is what the second benchmarks does. The results are displayed in Figure B.2.

Also these results show that the implemented map provides a good speedup compared to using the Java `HashMap` for this type of applications.

B.3.2 Index map

The index map maps one key to one or more integers, compared to the key map which maps it to only one integer. When inserts take place and the key already has a mapping, the new row index is appended to the value. Index maps are used for grouping queries and the performance of the index map is crucial for the query performance whenever groupings are present.

The index map implementation has the same advantage as the key map over the Java `HashMap`, it can hash and insert all the key columns without having to build a vector for each row and hash it. Since index maps only allow for one insert and can not grow, the benchmarking is relatively straightforward.

The first benchmark was run with non-overlapping data so that each insert will create a new mapping. The results are displayed in Figure B.3. The implemented map performs better in the whole range. There are however a few places that seem not hash

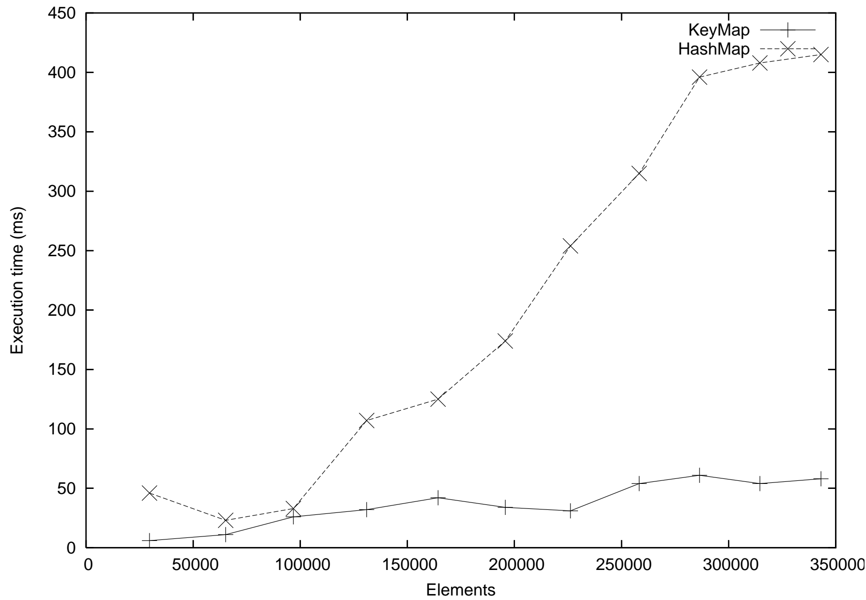


Figure B.2: Benchmark of the implemented `KeyMap` vs. Java `HashMap` with 50% overlapping data, i.e. 50% updates.

poorly, causing many collisions. This benchmark was run with the quote data using time and symbol columns as keys. The times are averaged from three independent runs.

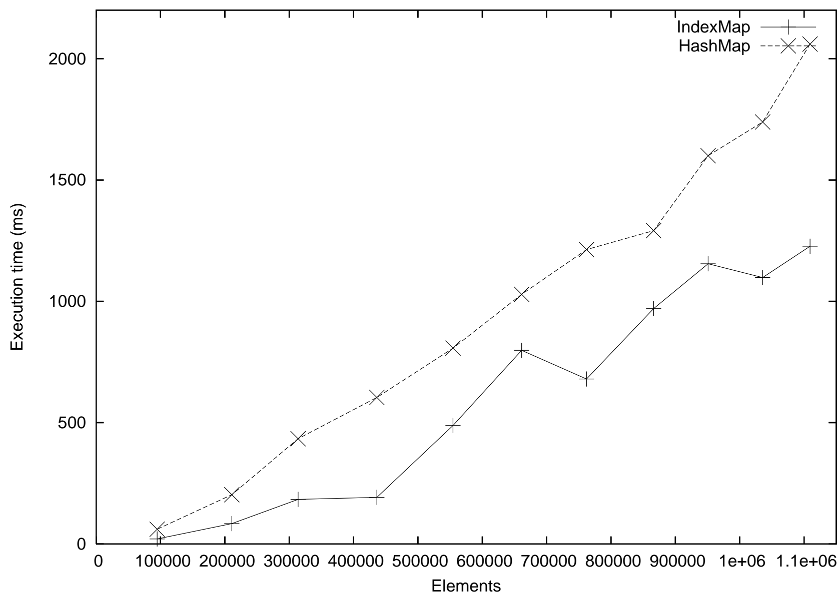


Figure B.3: Benchmark of the implemented `IndexMap` vs. Java `HashMap` with non-overlapping data, i.e. no updates, and two key columns.

The second benchmark uses the quote data as well but the times have been rounded to even seconds. This creates an average of about two values per key and gives approximately a 50% overlap in the data. The results are presented in Figure B.4. Also in this benchmark the implemented map performs much better than the Java map, in many cases the execution time is about half.

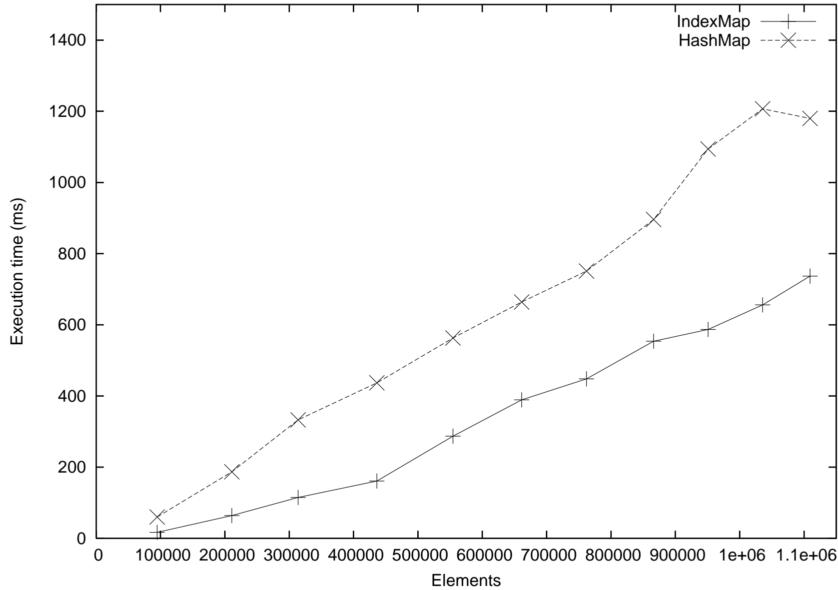


Figure B.4: Benchmark of the implemented `IndexMap` vs. Java `HashMap` with about two values per key (50% overlap) over two key columns.

B.4 Database benchmarks

This section covers query benchmarks performed in the database and compared to Q and PostgreSQL. The benchmark use the quote data and reflects typical database workloads at Nomura. First the different benchmarks are introduced and then the result is presented.

B.4.1 Query with condition

This query is a selection with a simple condition. The query selects rows from the quote data within a specific time interval and where the symbol is within a specified list of symbols. The symbol list is the following: ["HST", "FUA", "UOP", "EKW"], and the time interval is ['08:20:00.00', '08:55:00.000']. The query can be written as follows in SQL and Java:

Listing B.1: SQL for the condition query benchmark.

```
SELECT time, sym, bid, ask, asize, bsize FROM quote
    WHERE sym in ('HST', 'FUA', 'UOP', 'EKW')
        AND time > '08:20:00.000'
        AND time < '08:55:00.000'
```

Listing B.2: Java code for the condition query benchmark.

```
StringVector stocklist = new StringVector();
stocklist.append("HST");
stocklist.append("FUA");
stocklist.append("UOP");
stocklist.append("EKW");

Select sel = new Select(new Column[] {
    new Column("time"),
    new Column("sym"),
    new Column("bid"),
    new Column("ask"),
    new Column("asize"),
    new Column("bsize"),
  }, quote,
    new Condition[] {
        new In(new Column("sym"), stocklist),
        new Within(new Column("time"),
        new Atom(new TimeVector("08:20:00.000")),
        new Atom(new TimeVector("08:55:00.000")))},
    null
);
```

B.4.2 Simple aggregate

This benchmark uses simple aggregates over the columns in the quote data set. It selects the max values from the **date**, **time** and **ask** columns, and the sum of the **ask** and **bid** columns. The result table will contain only one row.

Listing B.3: Java code for the aggregate query benchmark.

```
Select sel = new Select(new Column[]{
        new Max("date"), new Max("time"),
        new Max("ask"), new Sum("ask"),
        new Sum("bid")
    },
    quote, null, null);
```

B.4.3 Grouped aggregates

This benchmark performs aggregate functions over grouped data. This means that the aggregate functions are applied over each group, resulting in one value per group. The query groups the data by symbol, date and minute and performs the aggregates over these groups. In pseudo SQL the query looks like the following:

Listing B.4: Pseudo SQL for benchmark of aggregates over grouped data.

```
SELECT date, time, sym, first(ask), first(bid), last(ask), last(bid),
       min(ask), min(bid), max(ask), max(bid), max (ask-bid), min(ask-bid),
       avg(ask-bid), count(*) from quote
    WHERE date within (startDate, endDate), time within (startTime,endTime),
          sym in stocklist
    GROUP BY date, sym, 60 second bins
```

B.4.4 Left outer join

The join benchmark joins the quote data with a table containing symbols and some other columns and which is keyed on symbol. The result will produce have one row for each row in the quote table. Symbols in the quote table that are not present in the other

table will contain null values for the new columns. The performance of this benchmark is very much up to the performance of the lookup method for the key map since this is used to look up the symbols in the keyed table.

In the Java code, a join can be performed using the primitive vector interface and will implicitly perform the join over the key columns in the *lookup*-table, i.e. the right hand side. In SQL the query looks as follows:

Listing B.5: SQL for left outer join benchmark.

```
SELECT * FROM quote LEFT JOIN lookup ON quote.sym = lookup.sym;
```

B.4.5 Results

The results of the benchmarks of the sample queries are displayed as plots in Figure B.5 and Figure B.6. Figure B.5 shows the results of PostgresSQL but in order to get some resolution the scale of the execution-time axis was limited. The execution time of the capped PostgreSQL column was over 17000ms. To get better resolution over the execution times of implementation and Q, a separate plot of these two is shown in Figure B.6.

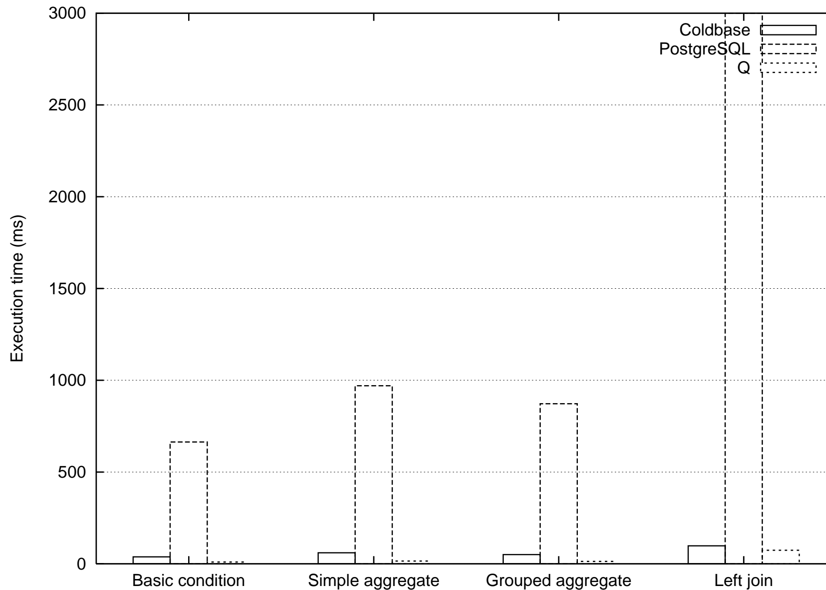


Figure B.5: Benchmark of the sample queries, including results from the implementation, PostgreSQL and Q.

B.5 Multi-threading benchmarks

The benchmarks in this section focus on multi-threaded tests.

B.5.1 Primitive vector operations

This benchmark executes vector operations in two threads simultaneously. The first thread computes the following:

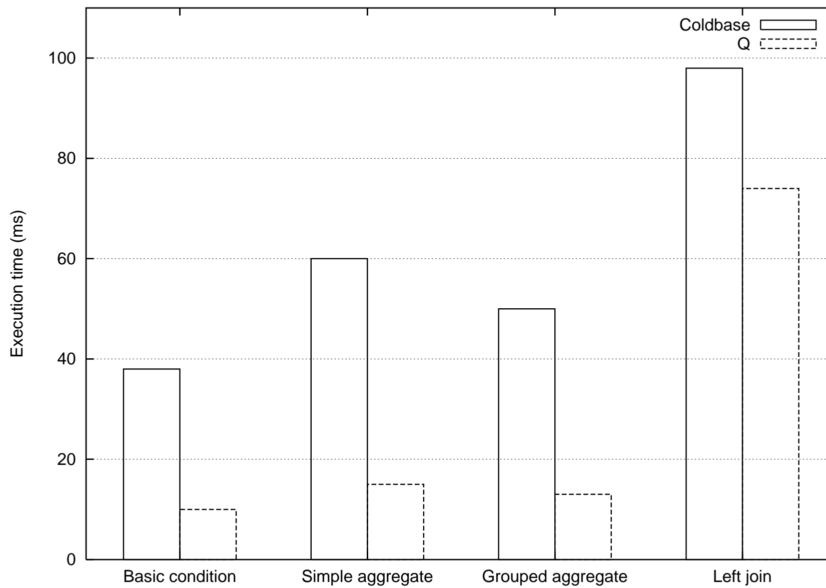


Figure B.6: Benchmark of the sample queries, including results from the implementation and Q.

Table B.3: Benchmark times in milliseconds of sample queries.

Benchmark	Java	PosgreSQL	Q
Basic condition	38	664	10
Simple aggregate	60	970	15
Grouped aggregate	50	872	13
Left join	98	17435	74

```
bid.mul(ask).add(bid).sqrt().sum();
```

and the second thread computes:

```
ask.mul(ask).div(bid).sqrt().max();
```

where `ask` and `bid` are the vectors of the *ask* and *bid* columns of the quote data. The computed value has no real meaning. The queries are first executed serially and then i parallel to establish how large the multi-threading gains are.

B.5.2 Aggregate query with groups

This test performs two identical aggregate queries with groupings and conditions. The query is first run by itself, and then using two threads in parallel. The corresponding query is SQL looks like this:

```
SELECT min(ask), min(bid), max(ask), max(bid),
       max(ask - bid), count(*)
FROM quote
```

```
WHERE sym in ("HST", "FUA", "UOP", "EKW", "FRJ", "NNT") AND
      time >= '08:10:00.000' AND
      time <= '08:50:00.000'
GROUP-BY date, sym, round(time, '00:01:00.000');
```

B.5.3 Another aggregate query with groups

This test performs two identical aggregate queries. This version uses more complex arithmetic and more CPU-intensive calculations.

```
SELECT min(ask), min(bid), max(ask), max(bid),
       max(ask - bid), count(*)
FROM quote
WHERE sym in ("HST", "FUA", "UOP", "EKW", "FRJ", "NNT") AND
      time >= '08:10:00.000' AND
      time <= '08:50:00.000'
GROUP-BY date, sym, round(time, '00:01:00.000');
```

B.5.4 Left outer join

In this benchmark two joins are performed first sequentially and then in parallel. The join is the same as the join benchmark in Section B.4.4.

B.5.5 Results

In Figure B.8 the execution times when run using one thread and when using two threads are displayed for the Intel multi-core system. The same benchmark was also run on the Sun UltraSparc machine and those results are displayed in Figure B.7.

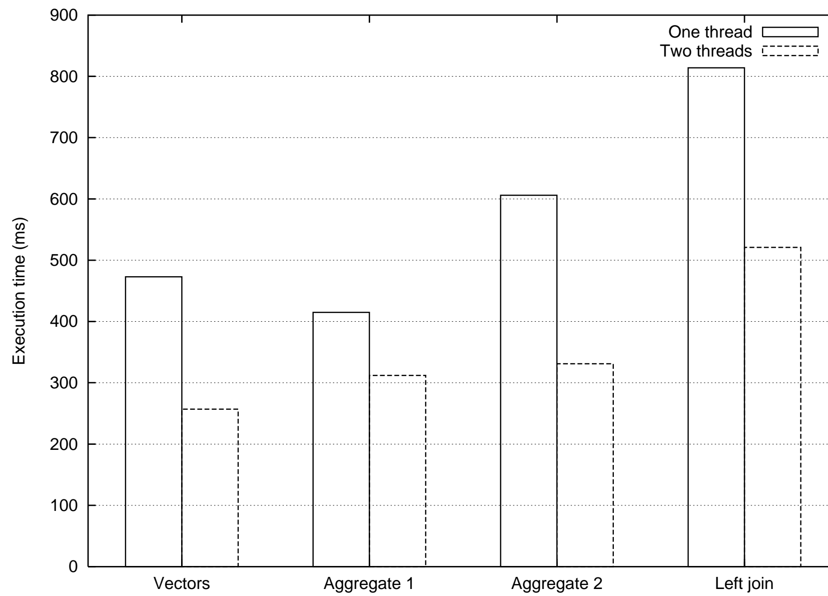


Figure B.7: Multi-threading benchmark displaying execution times using one and two threads, run on Sun UltraSparc IIIi.

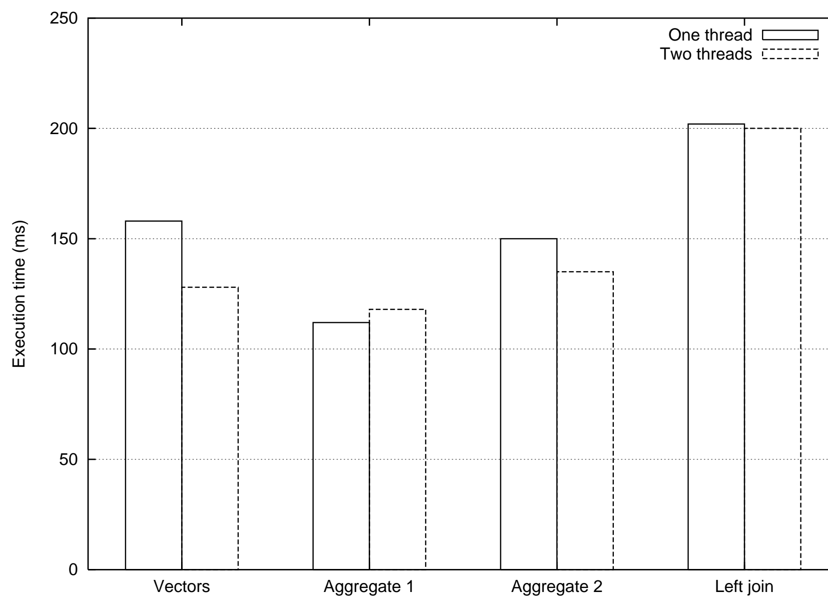


Figure B.8: Multi-threading benchmark displaying execution times using one and two threads, run on Intel Core 2 Duo.

Appendix C

Source Code

This appendix includes a few interesting parts from the source code.

C.1 Bit vector `where`

This method converts a bit vector into an index vector. The indexes of all set bits are stored in an integer vector. This is described further in Section 4.1.8.

Listing C.1: Implementation of `where`.

```
public IntVector where() {

  int used = 0;
  //Start with one eight of the number of current bits, minimum 64
  int[] result = new int[64 + (fill/8)*64];

  //Loop over all bits
  for (int i = 0; i < length; i += 64) {
    //loop over current long
    long elem = array[i/64];
    if(elem != 0)
    {
      for(int k = 0; k < 64; k+=4) {
        switch((byte)(elem & 0x000f)) {
        case 0: // 0000
          break;
        case 1: // 0001
          result[used++] = i + k;
        break;
        case 2: // 0010
          result[used++] = i + k + 1;
          break;
        case 3: // 0011
          result[used++] = i + k;
          result[used++] = i + k + 1;
          break;
        case 4: // 0100
          result[used++] = i + k + 2;
          break;
        case 5: // 0101
          result[used++] = i + k;
          result[used++] = i + k + 2;
          break;
        case 6: // 0110
          result[used++] = i + k + 1;
          result[used++] = i + k + 2;
          break;
        case 7: // 0111
          result[used++] = i + k;
```

```
          result[used++] = i + k + 1;
          result[used++] = i + k + 2;
          break;
        case 8: // 1000
          result[used++] = i + k + 3;
          break;
        case 9: // 1001
          result[used++] = i + k;
          result[used++] = i + k + 3;
          break;
        case 10: // 1010
          result[used++] = i + k + 1;
          result[used++] = i + k + 3;
          break;
        case 11: // 1011
          result[used++] = i + k;
          result[used++] = i + k + 1;
          result[used++] = i + k + 3;
          break;
        case 12: // 1100
          result[used++] = i + k + 2;
          result[used++] = i + k + 3;
          break;
        case 13: // 1101
          result[used++] = i + k;
          result[used++] = i + k + 2;
          result[used++] = i + k + 3;
          break;
        case 14: // 1110
          result[used++] = i + k + 1;
          result[used++] = i + k + 2;
          result[used++] = i + k + 3;
          break;
        case 15: // 1111
          result[used++] = i + k;
          result[used++] = i + k + 1;
          result[used++] = i + k + 2;
          result[used++] = i + k + 3;
          break;
        default:
          //
        }
        elem >>>= 4;

      }
    }
    if(used + 64 >= result.length) {
      result = Arrays.copyOf(result, result.length * 2);
    }
  }

  return new IntVector(result, Math.min(length, used), true);
}
```

C.2 IntVector *less than* method, `lt`

The `lt`-method performs a element-wise comparison between two vectors, resulting in a bit vector representing the boolean results of each comparison. Below is an excerpt from how the comparison methods work and how they build the result bit vector. This method is very similar for all types of comparisons, regardless of the data types. The excerpt does shows the case where both vectors are of equal type, i.e. no type coercion.

Listing C.2: Excerpt from implementation of `lt` in `IntVector`.

```
if (type == getDataType()) {
  //Coerce into full length vectors if one of the operands is an atom
  int[] lhsArray = coerceList(rhs).array;
  int[] rhsArray = (int[]) rhs.coerceInt().coerceList(lhs).array;
```

```
  long[] res = new long[fill/64 + 1];
  int currentIndex = 0;
  int currentBit = 0;
  for (int i = 0; i < fill; ++i) {
    //Set the bit if the condition evaluates to true
    if (lhsArray[i] < rhsArray[i]) {
      res[currentIndex] |= 1l << currentBit;
    }
    currentBit++;
    if(currentBit == 64){
      currentBit = 0;
      currentIndex++;
    }
  }
  //Wrap the result array in a bit vector and return it
  return new BitVector(res, fill);
}
```