AN EVALUATION OF QUERY PROCESSING STRATEGIES FOR COLUMN STORES

MASTER THESIS

Stefan Eigenmann
ETH Zurich
stefane@ethz.ch

November 17, 2009 - May 16, 2010

Supervised by:
Prof. Andreas Schmidt
Prof. Donald Kossmann
Abstract

In this thesis, it is analyzed how to best combine Crescando with a column-oriented data base management system. Crescando is a distributed, row-wise relational table implementation designed to answer a large amount of queries and do many updates in predictable time. This thesis explores different strategies to adapt Crescando to a column store. To achieve this, the data partitioning is analyzed and different setups of an analytical cost model looking at dimensions like the scanning order, merging and materialization strategies are developed and compared with each other. With help of the Amadeus workload, that was also used in Crescando, a clear winning strategy is found.
Acknowledgments

First of all I want to thank Prof. Donald Kossmann for enabling me to do my master thesis on such an interesting topic. I got a much better insight to column stores and the low-level operations done in it. Special thanks go to my supervisor Prof. Andreas Schmidt for his guidance, participation and valuable feedbacks. At this place I also want to thank Tudor-Ioan Salomie for his participation and feedback. Last, but not least, I also want to thank my family for their patience and support during this thesis.
# Contents

1 Introduction .................................................. 9  
   1.1 Problem Statement ...................................... 9  
   1.2 Contribution ........................................... 9  
   1.3 Thesis Outline ......................................... 10  

2 Fundamentals ................................................. 11  
   2.1 Column Store ........................................... 11  
       2.1.1 Query Evaluation .................................. 12  
   2.2 Crescando ............................................... 12  
       2.2.1 Architecture ...................................... 12  
       2.2.2 Algorithm ........................................ 13  
   2.3 CPU Cache ............................................... 14  
       2.3.1 Access Patterns .................................... 15  

3 Column Store Clock Scan ................................... 19  
   3.1 Upper Bound For Number Of Queries .................... 19  
       3.1.1 Naïve Implementation ............................. 19  
       3.1.2 Clock Scan With Indexed Predicates ............. 21  
   3.2 From One to Many Queries .............................. 23  
       3.2.1 Inverse Position List ............................. 23  
       3.2.2 Clock Scan Variants ............................... 24  
       3.2.3 Partitioning ...................................... 25  
   3.3 Dimensions .............................................. 29  
       3.3.1 Column Order ..................................... 29  
       3.3.2 Merging ........................................... 29  
       3.3.3 Materialization ................................... 30  
       3.3.4 Scanning Order .................................... 31  

4 Cost Models .................................................. 33  
   4.1 Assumptions ............................................. 33  
   4.2 Introduction of Different Model Setups ............... 34  
       4.2.1 External Merging - Late Materialization ........ 34  
       4.2.2 Internal Merging - Late Materialization ........ 34  
       4.2.3 External Merging - Early Materialization ....... 35  
       4.2.4 Internal Merging - Early Materialization ....... 35  
   4.3 CPU-Time Model ......................................... 35  
       4.3.1 External Merging - Late Materialization ....... 35  
       4.3.2 Internal Merging - Late Materialization ....... 36
Chapter 1

Introduction

1.1 Problem Statement

Traditional database management systems (DBMS) use a row store as their storage layout. This means that a relational table is stored row-wise. Since there is a widening gap between the faster CPU and slower memory performance, we don’t want to always load the whole rows of a table if we are only interested in a single column. Therefore column-oriented DBMS are an interesting alternative to existing row store based relational DBMS like MySQL or Oracle. They store a table column-wise and are especially interesting for data warehouses and other read-mostly applications. Whenever we want to read or query only a few attributes of a large table, we profit from a column store. There have been a lot of comparisons of column stores and row stores [5] [9] [10] [11].

On the other hand in-memory DBMS are becoming more popular with the RAM getting cheaper and larger. Crescando [14] is a CPU bound distributed in-memory table implementation, designed to perform large number of queries and updates with guaranteed access latency. Therefore a new scanning algorithm Clock Scan is used.

Although Holloway et al. [11] claims that column stores perform better when a program is I/O bound and row stores win in a CPU bound environment, it should be examined how to best implement Clock Scan on a column store. If a column store implementation loses to the existing row store implementation, the insights gained can still be used for an implementation on disc.

1.2 Contribution

There are a lot of possibilities to adapt Clock Scan to a column store. A first decision is whether to partition the data horizontally or vertically among the threads. In a horizontal partitioning, a thread is responsible for the same workload as in the row store approach of Crescando, whereas a vertical partitioning leads to a completely new setup. Further my challenge is to compare different strategies and to devise the best approach for implementing Clock Scan on a
column store. For instance, the best order in which the columns are scanned as well as the best time and method to do the merging have to be found. For the different methods analytical cost models regarding the CPU time as well as the access time must be developed and then evaluated against each other. This models should help making decisions when implementing a column store version.

1.3 Thesis Outline

This thesis is organized as follows. Chapter 2 gives a short introduction to the fundamentals used in this thesis. We first summarize the main characteristics of a column store database management system, then introduce the idea behind Crescando and Clock Scan and finally talk about CPU cache. In Chapter 3 we discuss different approaches of column store Clock Scan realizations. In Chapter 4 we design an analytical cost model and define different setups for CPU computing time as well as for access time, based on cache misses. In Chapter 5 we evaluate the model with the Amadeus workload and compare the different setups to each other. The thesis concludes in Chapter 6 where the findings are summarized and future work is proposed.
Chapter 2

Fundamentals

2.1 Column Store

A column store is a storage layout for DBMS, where data is stored by columns unlike in most traditional DBMS, where data is stored by rows. Assume we have this table.

<table>
<thead>
<tr>
<th>StudentId</th>
<th>Firstname</th>
<th>Lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Joe</td>
<td>Miller</td>
</tr>
<tr>
<td>2</td>
<td>Phoebe</td>
<td>O`Brien</td>
</tr>
<tr>
<td>3</td>
<td>Lauren</td>
<td>Johnson</td>
</tr>
</tbody>
</table>

To store this table, the data of the two-dimensional table has to be serialized. In a row store, for each row all values of that row are serialized together: 1, Joe, Miller 2, Phoebe, O`Brien 3, Lauren, Johnson. In a column store, for each column all values of that column are serialized together: 1, 2, 3 Joe, Phoebe, Lauren Miller, O`Brien, Johnson.

Today there are quite a few column oriented DBMS implementations out there, e.g. C-Store by Stonebraker et al. [13] or the in-memory database MonetDB by Boncz et al. [7, 8]. Both claim to be faster than traditional DBMS for read-mostly data. A column store profits if we don’t have to touch every attribute in a table, but only a few, e.g. if we have projections on only a few columns or if we want to compute an aggregation on only a few columns. Both disk seek time can be reduced and cache locality improves when we store the data column-wise.
2.1.1 Query Evaluation

If we want to evaluate a query on a column store, we evaluate the predicates on the corresponding columns, get tuples \(<\text{position}, \text{value}>\) and then have to merge them by their positions to generate row-wise tuples. There are two strategies that are distinguished: early and late materialization. While in early materialization we merge the results as early as possible to generate row-wise tuples on which the further steps of the query plans are executed (analog to a row store), in late materialization we keep the column representation as long as possible. Abadi et al. analyzed these two strategies in [6] and didn’t find a clear winner. If output data is aggregated, we have highly selective predicates or the input data is compressed, late materialization should be used and otherwise early materialization performs better.

2.2 Crescando

The goal of Crescando [14] was to design a technology to answer a huge amount of unpredictable queries and updates within a predictable time for a relational table. Traditional database systems are designed to optimize the performance of every individual query and not to achieve highest throughput or lowest latency for an unpredictable query workload.

Crescando was motivated by Amadeus, a world-leading service provider for managing travel-related bookings. The bookings, that are kept on-line, are stored in a single flat table of a size of several hundred gigabytes and that table sustains a workload of several hundred updates and several thousand queries per second. Since the number of queries that select on non-key attributes is increasing and the system had reached a level of complexity where adding views and indexes was no longer feasible, many queries had to be answered in batch (off-line). Therefore Crescando was proposed.

2.2.1 Architecture

Crescando is a distributed architecture and fully runs in main memory. However, the entire table might not fit in the main memory of a single machine. So, for scalability and availability, the data is horizontally partitioned and replicated among different machines (storage nodes). In the following thesis, the investigations are limited to a single non-replicated storage node.

The architecture of a storage node can be seen in Figure 2.1. There are two main functions to the outside: enqueue operations (queries and updates) to the input queue and dequeue results form the output queue. Inside the architecture the operations are split and distributed among scan threads. Each scan thread is a kernel thread with hard processor affinity, that continuously scans a horizontal partition of the data and outputs the result tuples (Section 2.2.2). In the end, if an operation was split to several scan threads, the results get merged. A scan thread can handle thousands of operations together and due to the nature of the algorithm, the execution takes roughly the same time for all operations.
2.2.2 Algorithm

The algorithm used in the scan threads is called Clock Scan. A Clock Scan iterates over all records of its data partition and performs a query/update-data join over a set of operations. In a query-data join the queries are looked at as a relation of predicates. In the following, we restrict ourselves to those queries, that have a selection predicate, which can be expressed as a conjunction of predicates of the form $\text{attribute op const}$, $\text{op}$ being a comparison operator. Equality predicates of the form $\text{attribute} = \text{const}$ are represented as a relation $Q$ with heading $(\text{qid}, \text{const})$, $\text{qid}$ being the query id. If we want the union of the results over a data relation $R$, we can perform the join $Q \bowtie \triangleleft \text{attribute} = \text{const} \ R$. Similarly range predicates of the form $\text{lb} \leq \text{attribute} \leq \text{ub}$ are represented as $(\text{qid}, \text{lb}, \text{ub})$.

The query-data join is implemented as an index union join, where an index is built over query predicates. This is in contrast to traditional database systems, where the data in the table is indexed. So, every query has a predicate, that is part of an index, or the query is part of the set of unindexed queries. For every data record, probing the predicate indexes takes irrelevant queries out of consideration. Matching queries as well as unindexed queries have to be executed on the record.

Hence a scan thread does the following work.

- Take the queries from its input queue
- Iterate over the data relation $R$
- Join each record with the set of queries and the set of updates
- Output the results to its corresponding output queue

The algorithm of Crescando Clock Scan is CPU bound. This means that most of the time in the algorithm is spent in the CPU doing computations and the time to complete a task is mainly determined by the speed of the CPU. This is in contrast to an I/O bound algorithm, in which most of the time is spent doing input and output operations to main memory or disc.
2.3 CPU Cache

CPU cache is an aid to improve performance by storing copies of the data from recently or frequently used main memory locations, close to the CPU, such that the distance and hence the access time decrease. So cache is a smaller, faster memory. CPU cache often consists of several levels of cache: L1, L2, etc. Since cache is very expensive compared to main memory, the size of the cache is much smaller. The higher the number of the cache level, the slower the access to it becomes, the cheaper it gets and the bigger the cache becomes. Each cache level consists of many cache lines. The size and number of these cache lines differ from CPU to CPU and from cache level to cache level. Whenever we access some data, not only this data, but also the data stored close to it is loaded: only full cache lines are loaded. Since we often want to access this data as well anyway, this results in a big advantage. If we want to access some data, we speak of a cache hit if the requested data is already in the cache, and of a cache miss if the data has to be loaded from its original storage, respectively.

Because the cache size is limited, we have to decide what data to flush from the cache, whenever we want to load data in an already full cache. There is a big number of different cache replacement algorithms, like LRU (least recently used), MRU (most recently used) or LFU (least frequently used) to name just a few. In this thesis I always assume, that we have a LRU cache policy, i.e. whenever there is no space in cache anymore and an item has to be discarded, the one least recently used is chosen.

Cache associativity [1] is a term to describe where a location in main memory can be stored in cache. The two extreme cases are direct mapped cache and fully associative cache. In a direct mapped cache, each location in main memory can be cached by just one cache location, leading to the best cache hit times, but more cache misses. In a fully associative cache, each location in main memory can be cached by any arbitrary cache location, leading to less cache misses, but a longer time to find data in cache at cache hits. In the following we assume that we have a fully associative cache, i.e. that any data can be put everywhere in the cache.

Further we assume to have an inclusive cache [2]. That means that all data in the L1 cache is also in the L2 cache. We also assume that writing to an address in memory doesn’t result in any cache misses, i.e. the cache is fully bypassed.

As a basis for the model of the access times of my theoretical cost analysis I used the dissertation of S. Manegold [12]. So all the symbols and basic formulas in Section 2.3.1 are adopted from his work.

We assume the cache access latency for an access in the L1 cache to be part of the CPU cost as it can’t be avoided. The total access cost can then be expressed as the sum of the time to read a data from the L2 cache when not in the L1 cache plus the time to read it from memory if it’s not in the L2 cache, etc if there are further cache levels. So the access time is:

$$ T_{Acc} = M_{L1} \cdot \lambda_{L2} + M_{L2} \cdot \lambda_{Mem} $$  \hspace{1cm} (2.1)

where \( M_{Li} \) denotes the number of cache misses in the \( i^{th} \) cache level and \( \lambda_x \)
denotes the cache/memory access latency for the according level.
To analyze a given algorithm with respect to its access time we therefore have to
look at the number of cache misses. This approach is dependent on parameters
like the cache size or the cache line length.

2.3.1 Access Patterns

There is an infinite number of different possible algorithms. Finding the cache
misses for an arbitrary algorithm is tedious and error-prone. To handle this
problem, Manegold designed a model where each algorithm can be expressed as
a combination of basic access patterns, executed in sequence or in parallel. He
derived the cache misses for these basic access patterns and defined rules how
to combine them.
In this section first all the symbols used are listed, before the basic access
patterns are introduced. As I only introduce the parts of the model that are
required for the analysis of different Clock Scan implementations, the interested
reader is referred to [12] for derivations of the formulas as well as a detailed
description of the full model.

Symbols

Figure 2.2: Symbols

The number of cache misses of an access pattern \( P \) is denoted with \( M(P) \).
\( R \) is a data block and \( |R| \) is the number of data items (rows) in \( R \). To describe
the size in main memory (in bytes) of one data item, \( \bar{R} \) is used (we assume data
items to have fixed widths within a data block) and so the total size of a data
block $R$ can be expressed with $\|R\| := |R| \cdot \frac{z}{R_z}$.

# is the number of cache lines and $z$ is the size of one cache line (in bytes). The number of cache lines covered by $R$ is defined as $|R_z| := \lceil \|R\|/z \rceil$.

The symbols are illustrated in Figure 2.2.

### Basic Access Patterns

#### Single Sequential Traversal

The simplest access pattern is a single sequential traversal over $R$: \texttt{s\_trav(R[u])}. $u$ is an optional parameter and denotes the number of bytes read from each data item. If not specified, $u$ is assumed to be $R$, i.e. the whole row is read. To determine the number of cache misses, we have to distinguish two cases:

- **Case $R - u < z$** (the number of unread bytes is smaller than a cache line size, Figure 2.3):
  \[
  M(\texttt{s\_trav}(R, u)) = |R_z|
  \]

  ![Figure 2.3](image)

- **Case $R - u \geq z$** (the number of unread bytes is larger than a cache line size, Figure 2.4):
  \[
  M(\texttt{s\_trav}(R, u)) = |R| \cdot \left( \left\lfloor \frac{u}{z} \right\rfloor + \frac{(u - 1) \mod z}{z} \right)
  \]

  ![Figure 2.4](image)

#### Repetitive Sequential Traversal

\texttt{rs\_trav(n, dir, R[u])} is a sequential traversal that sweeps $n$ times over $R$. The parameter \textit{dir} is \texttt{dir = uni} if the traversals are uni-directional and \texttt{dir = bi} if they are bi-directional, respectively. If we always traverse $R$ in the same direction and $R$ fits into the cache, we have the same number of cache misses as for a single sequential traversal. However if $R$ doesn’t fit into the cache, because of the LRU replacement policy, every data item that is read again was already replaced by another item, resulting in $n$ times as much cache misses for $n$ traversals.

\[
M(\texttt{rs\_trav}(n, uni, R, u)) = \begin{cases} 
M(\texttt{s\_trav}(R, u)) & \text{if } M(\texttt{s\_trav}(R, u)) \leq \# \\
R \cdot M(\texttt{s\_trav}(R, u)) & \text{if } M(\texttt{s\_trav}(R, u)) > \#
\end{cases}
\]
Random Access  In a random access pattern \( r_{\text{acc}}(r,R,[u]) \) there are \( r \) consecutive, independent random accesses on \( R \). A data item may be hit more than once and not every item has to be hit. With \( \mathcal{P} = r_{\text{acc}}(r,R,u) \) the number of cache misses is

\[
M(r_{\text{acc}}(r,R,u)) = \begin{cases} 
C(\mathcal{P}) & \text{if } C(\mathcal{P}) \leq \# \\
C(\mathcal{P}) + \left( \frac{r}{I(\mathcal{P})} - 1 \right) \cdot \left( C(\mathcal{P}) - \frac{\#}{C(\mathcal{P})} \cdot \# \right) & \text{if } C(\mathcal{P}) > \#
\end{cases}
\]

where \( C(\mathcal{P}) \) is the number of distinct cache lines touched and \( I(\mathcal{P}) \) the number of distinct data items accessed.

Compound Access Patterns

When combining basic access patterns to more complex ones, we can’t simply sum up the cache misses of the basic patterns because of cache interference.

Sequential Execution  We are looking at a sequential execution of \( n \) access patterns \( (\mathcal{P}_1 \oplus \mathcal{P}_2 \oplus \cdots \oplus \mathcal{P}_n) \) where \( \mathcal{P}_{i+1} \) is executed after \( \mathcal{P}_i \) is finished. Hence the access patterns \( \mathcal{P}_i \) do not interfere. When a pattern uses the same data as the pattern before, it can benefit from the data left in cache. Otherwise the cache misses of the sequential access patterns \( \mathcal{P}_i \) can just be summed up.

Parallel Execution  For a parallel execution of \( n \) access patterns \( (\mathcal{P}_1 \odot \mathcal{P}_2 \odot \cdots \odot \mathcal{P}_n) \) we don’t have the full cache available for every access pattern, but only a fraction of the number of cache lines, instead. So we have to replace every \( \# \) by \( \# / v \) where \( v \) for an access pattern \( \mathcal{P}_q \) is defined as

\[
v_q = \frac{F(\mathcal{P}_1 \odot \cdots \odot \mathcal{P}_n)}{F(\mathcal{P}_q)}
\]

and the footprint size \( F \) as

\[
F(\mathcal{P}) = \begin{cases} 
1 & \text{if } \mathcal{P} = s_{\text{straw}} \\
|R|_{z} & \text{else.}
\end{cases}
\]

Further is

\[
F(\mathcal{P}_1 \odot \cdots \odot \mathcal{P}_n) = \sum_{i=1}^{n} F(\mathcal{P}_i)
\]

and

\[
F(\mathcal{P}_1 \oplus \cdots \oplus \mathcal{P}_n) = \max\{F(\mathcal{P}_1), \ldots, F(\mathcal{P}_n)\}.
\]
Chapter 3

Column Store Clock Scan

Crescando (Section 2.2) handles both, query and update operations. In this thesis we are looking at an adaption to a column store, where we limit the analysis to a read-only workload, though. Like Crescando, we also only look at queries with a selection predicate, that can be expressed as a conjunction of predicates.

In this section we first find an upper bound for the number of queries, that can be evaluated together, then discuss different approaches of column store Clock Scan realizations and finally introduce the dimensions, that are distinguished in the model of chapter 4.

3.1 Upper Bound For Number Of Queries

For performance reasons we want the inner most part of a Clock Scan to fit in the L1 cache. Therefore we deduce a formula for the maximum number of queries that may be put into a Clock Scan together, such that a predicate index fits into the L1 cache.

3.1.1 Naive Implementation

As a first simple task we analyze an indexless naive implementation of Clock Scan over a single data block $R$: While a Clock Scan can be looked at as a cursor, that sweeps over $R$, at each data item a second cursor traverses through all the predicates $P$ and outputs the matching ones. Hence this simple version of clock scan can be seen as the following compound access pattern $\mathcal{P}$:

$$\mathcal{P} = s_{\text{trav}}(R, u_R) \odot r_{s_{\text{trav}}(|R|, \text{uni}, P, u_P})$$

To determine the number of cache misses of $\mathcal{P}$, first the footprint sizes of its parts have to be looked at: $F(s_{\text{trav}}(R, u_R)) = 1$ and $F(r_{s_{\text{trav}}(|R|, \text{uni}, P, u_P)}) = |P|z$. Since the number of cache misses of a single sequential traversal is independent of the number of cache lines $\#$ we only have to determine the fraction $v$ of
the dedicated cache for the repetitive sequential traversal \(rs_{trav}(R, uni, P, u_P)\): 

\[v_{rs_{trav}} = \frac{|P|+1}{|P|} \cdot |P|\] 

So 

\[M_{NaiveCS} = M(rs_{trav}(R, uni, P, u_P)) + M(rs_{trav}(u_P, uni, P, P))\] 

\[M(s_{trav}(P, u_P)) \leq \frac{|P|}{|P|+1} \cdot \# \] 

(3.1)

To obtain highest performance by not suffering from cache misses caused by query predicates we want: 

\[M(s_{trav}(P, u_P)) \leq \frac{|P|}{|P|+1} \cdot \# \]

Case \(u_P < z\):

\[|P| \leq \frac{|P|}{|P|+1} \cdot \# \]

\[\Leftrightarrow 1 \leq \frac{1}{|P|+1} \cdot \# \]

\[\Leftrightarrow |P| \leq \# - 1 \]

\[\Leftrightarrow |P| \cdot \frac{T}{z} \leq \# - 1 \]

\[\Leftrightarrow |P| \leq \frac{z \cdot (\# - 1)}{T} \]

Case \(u_P \geq z\):

\[|P| \cdot \left(\frac{|u_P|}{z} + \frac{(u_P - 1) \mod z}{z}\right) \leq \frac{|P|}{|P|+1} \cdot \# \]

Substitute \(a := \left(\frac{|u_P|}{z} + \frac{(u_P - 1) \mod z}{z}\right)\)

\[|P| \cdot a \leq \frac{|P|}{|P|+1} \cdot \# \]

\[\Leftrightarrow |P| \cdot \frac{T}{z} \cdot \# \leq \frac{|P|}{|P|+1} \cdot \# \]

\[\Leftrightarrow \left(\frac{|P| \cdot T}{z} + 1\right) \leq \frac{|P| \cdot T}{z} \cdot \# \]

\[\Leftrightarrow |P| \leq \frac{\# - 2 \cdot \frac{z}{T}}{a} \]

By resubstituting \(a\) we get

\[|P| \leq \frac{|u_P|}{z} + \frac{(u_P - 1) \mod z}{z} - 2 \cdot \frac{z}{T} \]

The latter case is just for completion and probably never comes true since we are always interested in the whole predicate. It may become interesting if we
want to look at Parallel n-Column Clock Scans though (Section 3.2.2). With this formula we now can calculate how many predicates we may put into our clock scan system and how many cache misses we can expect. Therefore we assume the following numbers regarding the L1-cache: \( z = 64 \) byte and \( \# = 1024 \). Assuming we have range predicates of the form \((qid, lb, ub)\) where \( qid \) is the query id the predicate belongs to (fitting in 2 byte) and \( lb \) and \( ub \) are the lower- and upper bound of the attributes (represented as integers), \( P = 10 \) byte. So the number of predicates \(|P|\) may maximally be \( 64\text{byte} \cdot 1023/10\text{byte} = 6547 \).

### 3.1.2 Clock Scan With Indexed Predicates

However, because we want the matching queries to be found very fast, we build indexes over the predicates, like Crescendo did.

So a better implementation of Clock Scan has the form

\[
P = s_{trav}(R, u_R) \odot r_{acc}(|R|, I, u_I)
\]

where \( I \) denotes the index over the predicates. Again we first look at the footprint sizes of its parts: \( F(s_{trav}(R, u_R)) = 1 \) and \( F(r_{acc}(|R|, I, u_I)) = |I|_z \).

As before the fraction \( v_{s_{trav}} \) of the dedicated cache for the repetitive sequential traversal we now have to determine \( v \) for the cache available for our index: \( v_{r_{acc}} = \frac{|I|_z + 1}{|I|_z} \).

\[
M_{IndexedCS} = M(s_{trav}(R, u_R)) + M(r_{acc}(|R|, I, u_I))
\]

\[
C(P) \text{ if } C(P) \leq \frac{|I|_z}{|I|_z + \#}
\]

To determine the number \( C \) of distinct cache lines touched we have to look at how the index is built.

### Hash Index

If we only have equality predicates of the form \((qid, const)\) on an attribute, we use a hash map as index. In a hash index for an attribute \( a \), every query id of all queries, that have a predicate on \( a \), is stored. Further also every different value appearing in a query predicate is stored in the hash map. In worst case, every query has a predicate on \( a \) and every predicate targets a different value. So we have a hash map with \(|P|\) different keys and \(|P|\) different qids as entries and hence the size \(|I|\) of a hash index is roughly \(|P| \cdot \frac{P}{z} = |P|\).

In worst case, we have to load the whole index into the cache and therefore touch \( C = |I|_z = |P|_z \) different cache lines.

We get the same requirement as before:

\[
|P| \leq \frac{z \cdot (\# - 1)}{P}
\]
Packed 1D R-Tree

As an index for the range queries we can use a one-dimensional R-Tree stored in an array. The leaves of the tree store the predicates. For a fanout of 2 the children of a node at offset $i$ can be found at offsets $2i + 1$ and $2i + 2$, and generally for a fanout $f$ at offsets $f \cdot i + j \forall j \in \{1, \ldots, f\}$. The tree is constructed by first sorting the ranges by midpoints ($(lb + ub)/2$) and then building the tree bottom up from left to right storing in each intermediary node a new range with lower- and upper bounds as the minimal lower- and maximal upper bound of its children: $lb = \min\{lb_1, \ldots, lb_f\}, \quad ub = \max\{ub_1, \ldots, lb_f\}$. This leads to a tree in which every node’s left outest subtree is complete. The queries matching a given value can be found by checking the value with the ranges of every child and then, if matched, recursively going down the tree.

We want to find the length $|I|$ of the array used to store the r-tree dependent on the number $|P|$ of predicates indexed and the tree’s fanout $f$. For a fanout 2, the height of the tree is $H = \lceil \log |P| \rceil$. We assume that we always store a complete tree, even though the unused leaves on the lowest level would not have to be stored as empty entries in the array, resulting in a shorter array. In a complete tree on the lowest level there are $2^{\lceil \log |P| \rceil}$ leaves. On the 2nd lowest level there are $2^{\lceil \log |P| \rceil}/2 = 2^{\lceil \log |P| \rceil}-1$ nodes and on the $i$th lowest level there are $2^{\lceil \log |P| \rceil} - (i-1)$ nodes. Since there are $H + 1 = \lceil \log |P| \rceil + 1$ levels in the tree, the total number of nodes in a complete tree is $\sum_{i=0}^{\lceil \log |P| \rceil} 2^{\lceil \log |P| \rceil} - i = 2 \cdot 2^{\lceil \log |P| \rceil} - 1$ (and generally for a fanout $f$: $f \cdot f^{\lceil \log f |P| \rceil} - 1$).

Determining the number $C$ of distinct cache lines touched in Equation 3.2 is not easy since there are empty parts in the index-array that are never read and hence, if large enough spaces, whole cache lines that are never touched. To simplify this we assume that the whole index array is loaded into the cache, i.e. $C = |I|_z$. The real value would probably be a little smaller.

Not all entries in the array have the same width. An intermediate node of the tree has the form $(lb, ub)$, whereas a leaf has the form $(qid, lb, ub)$. To simplify the calculations, we pessimistically assume every entry to have the form $(qid, lb, ub)$. Hence, the width of a data item in the array is $I = P$. With this assumption we get the requirement

$$|I| \leq \frac{z \cdot (\# - 1)}{P}$$

where $|I| = 2 \cdot 2^{\lceil \log |P| \rceil} - 1$ for a fanout 2. With the same assumptions as in Section 3.1.1 ($z = 64$ byte, $\# = 1024, P = 10$ byte) we get $|I| \leq 6547$ and hence $|P| \leq 2048$ for a fanout 2 (and $|P| \leq 2187$ for a fanout 3 and $|P| \leq 4096$ for a fanout 4).
3.2 From One to Many Queries

3.2.1 Inverse Position List

If we had a single query, the merging of the position lists in a Clock Scan would be the same as in a simple column store (Section 2.1) and there would not be a problem. However we are dealing with a few thousand queries at the same time in a Clock Scan. If we want to instantly merge the position lists (see Internal Merging in Section 3.3.2) after the output of a column scan, updating that many position lists would become too tedious. Therefore we introduce the concept of an inverse position list.

In an inverse position list we have for every position a list of all the query ids that output this position as a result. Because such a list can be huge (at least in the beginning after only one predicate execution) we store a query id list as a bitmap. Such a bitmap has length (in bit) equal to the number of queries we evaluate together. If we have an inverse position list \(<pos, \text{bitmap}>\), a 1 at position \(i\) in the bitmap denotes that the attribute value at position \(pos\) matches the predicate with query id \(i\). After a query index probing, the bitmap for this position can then be merged. The merging is easy and fast since we can do a bit-wise AND for conjunctions (and a bit-wise OR for disjunctions of query predicates).

We illustrate this with help of an example (see Figure 3.1). We have a column \(a\) with some integer values and 4 queries: Q1, Q2, Q3 and Q4, that have a predicate on \(a\). Traditionally we now would store a position list for every query (see the Traditional Position Lists in the example). Now, if a Clock Scan scans through a second column \(b\) and probes the first item of \(b\) against the corresponding predicate index, we would have to go through every query’s position list to update them. Therefore we don’t store a position list for every query, but a list of the query ids for every position (Inverse Position List in example). Therefore after the probing of the first item of column \(b\) we can simply merging the result with the query list for position 0.

<table>
<thead>
<tr>
<th>Queries:</th>
<th>Traditional Position Lists:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1: ...WHERE a = 9 ...</td>
<td>Q1: 0</td>
</tr>
<tr>
<td>Q2: ...WHERE a &lt; 20 ...</td>
<td>Q2: 0, 1, 2</td>
</tr>
<tr>
<td>Q3: ...WHERE a &gt; 27 ...</td>
<td>Q3: 3</td>
</tr>
<tr>
<td>Q4: ...WHERE a &gt; 5 ...</td>
<td>Q4: 0, 2, 3</td>
</tr>
</tbody>
</table>

| 9 | 0: Q1, Q2, Q4 | 0: 1101 |
| 4 | 1: Q2 | 1: 0100 |
| 18 | 2: Q2, Q4 | 2: 0101 |
| 43 | 3: Q3, Q4 | 3: 0011 |

Column a | Inverse Position List | Inverse Position List Using Bitmaps

Figure 3.1: Example - Inverse Position List
Optimization

Since this bitmap is still quite big and has to be loaded again and again, we introduce a few optimization techniques, all with the goal to reduce the amount of data, that has to be loaded.

One optimization is that we store an extra bit (and-ing every bit of the bitmap), that indicates us whether we have to load the whole bitmap for a position. So, if the whole bitmap would be 0, we wouldn’t have to load it. However, the bitmap for a position would be loaded for every column even if the only 1s in the bitmap are of queries, that are only interested in one or two columns. With a query being interested in a column, we mean, that that query has a predicate or a projection on that column. So we can also consider to not only keeping an overall extra bit, but to keep an extra bit for each column (and-ing bits of every query, that is interested in that column). For a position in every column, we then can look at the corresponding bit to know if we have to load the whole bitmap for that position or not. While this approach benefits from loading the bitmaps much less often, it suffers from a harder and more costly maintenance of these extra bits.

An other optimization is to compress the position lists. Naturally a low selectivity (i.e. highly selective predicates) leads to better compression. Ideally we can do the merging directly on the compressed data, e.g. when we use run-length-encoding. The best compression algorithm has yet to be determined. It should have a high compression as well as a good performance to do the merging. These parameters depend on the distribution of the values and the selectivity (very highly selective predicates naturally lead to better compression). Interesting in this perspective is certainly the paper “Integrating Compression and Execution in Column-Oriented Database Systems” by Daniel Abadi et al. [4]. They introduce compression on column stores and show how certain operations can be executed directly on compressed data.

3.2.2 Clock Scan Variants

In a column store, there are different possibilities of Clock Scans:

- Simple Clock Scan
- Parallel n-Column Clock Scan
- Sequential n-Column Clock Scan

A simple Clock Scan iterates over a single column and hence evaluates one attribute. An n-column clock scan on the other hand is a cyclic iteration over n columns. In a parallel n-column Clock Scan these n columns are stored row-wise, i.e. we don’t have a pure column store in this case. This parallel n-column Clock Scan is useful if we have many queries with the same n attribute predicates. Such useful conditions are for example: 'a = 10 AND b = 20' or 'a >18 AND b = 42'. In this case we can evaluate n attributes together, which leads to less queries being selected. It also becomes especially useful if we have compound primary keys. In a sequential n-column Clock Scan the n columns are stored column-wise
and traversed sequentially and therefore we can only evaluate one attribute at a time.

There exist several possible outputs for all variants of clock scan. Either a list of the matching positions is returned or the positions together with the value(s). In the latter case we may output them as a list of tuples \langle position, value(s) \rangle or as two separate lists positions and values. If we have late materialization (Section 3.3.3), we are only interested in the list of the matching positions, whereas in early materialization we also want the values.

### 3.2.3 Partitioning

In contrast to a row store Clock Scan, on a column store Clock Scan there are different possibilities to partition the data among the scan threads. As in Crescendo we want to have one thread per data fragment that runs in its own core.

**Vertical Partitioning**

One possibility is to partition the data vertically among the different scan threads (figure 3.3 left), such that each thread is responsible for a single column.

**Global Scheduler**

We have a global scheduler, that analyzes the query workload and then sets up an instance of a Clock Scan for every scan thread. We now can have fix, variable or hybrid column-to-thread-assignments. In a fix
case a thread is strongly coupled with one or multiple columns. If a thread is responsible for columns A and B, it may scan them together or alternately (see Section 3.2.2) and if it is responsible for even more columns, even more complicated patterns are possible. Assume column A appears more often than B and C. Now the scanning order can be A,B,A,C,A,B,A,C.

In a variable scenario the global scheduler looks at the query workload and then distributes the columns between the different scan threads.

In a hybrid setup we have both, fix and variable assignments: if an attribute appears nearly in every query predicates, there is a thread that always scans through this column. Less frequent columns are assigned variable.

Since a thread always scans through the same column(s), we can also input new queries at any time and not only in the beginning of a new scan run at position 0.

Query Scheduler In this approach, we also have a query scheduler. Its task is to

- take the queries from the input queue one at a time
- make an execution plan
- input the queries to the different scan threads according to the execution plan
- collect the results of the scan threads, merge them and build the final result
- put the results on the output queue

There are many different possible evaluation plans for a query, depending among other things on whether the output of one Clock Scan is given as input to the next one or not. In Figure 3.4 we see an example of a query plan where the output of a Clock Scan is given to the next Clock Scan and hence the scan threads have to be executed in sequence, whereas in Figure 3.5 we see an example of a query plan where the Clock Scans on the predicates run in parallel an
Figure 3.4: Position List as Input, Scan Threads on Predicates Executed Sequentially

Figure 3.5: Scan Threads On Predicates Executed in Parallel, Position Lists Merged in the End
the outputs are merged in the end.

While this partitioning would allow us to widen the workload to several relational tables, it is not clear how to best realize this query scheduler. The co-operation between the different scan threads is really hard and if we later want to allow updates too, consistency is hard to guarantee.

**Horizontal Partitioning**

The data can also be partitioned horizontally among the different scan threads (Figure 3.3 right), analog to the row store Crescando. So a thread’s task is to

- take queries from the input queue
- scan all columns for the rows it is responsible for and merge the outputs of the columns
- put results to the output queue

A query stays on the input queue till it is taken by every scan thread. For the merging, the best strategy is to use inverse position lists (Section 3.2.1). The scheduler’s work is to analyze the query workload, to build the predicate indexes and to determine the order in which the columns are scanned.

**Conclusion**

The different partitioning strategies lead to different implementations and problem statements. Both approaches have advantages and disadvantages.

**Vertical Partitioning**

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Easily extendable to multiple tables</td>
<td>- Consistency at updates hard to guarantee</td>
</tr>
<tr>
<td>- Queries can be added anytime</td>
<td>- Extremely complex (query scheduling and synchronization of different scan threads)</td>
</tr>
</tbody>
</table>
Horizontal Partitioning

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Easier for updates to maintain consistency, since only one thread touches a row (like in Crescendo)</td>
<td>- Thread’s work gets more complicated (merging, not only scanning)</td>
</tr>
<tr>
<td>- No extra query scheduler necessary</td>
<td></td>
</tr>
</tbody>
</table>

We have decided to further analyze the horizontal partitioning of the data among threads and skip the exact analysis of a vertical partitioning.

3.3 Dimensions

To find out how to best implement a column store Clock Scan we analyze four different dimensions and later develop an analytical cost model for each combination of them in Chapter 4.

3.3.1 Column Order

The order in which the columns are scanned has an impact on the number of rows that have to be probed on the following columns. Since we are only looking at conjunctions of predicates and assuming the scanning order of the columns is $C_1, \ldots, C_n$, we do not have to probe the value at a given position $p$ in a column $C_i$ ($i = 2 \ldots n$) if the value at position $p$ in $C_1$ does not match a query $q \in Q_i$, where $Q_i$ denotes the set of queries that have a predicate on $C_i$.

3.3.2 Merging

In a column store, if a query has predicates on several columns, the values of the matching positions have to be merged to generate the tuples that are returned (see Section 2.1).

Since in clock scan we are looking at several thousand queries in parallel, we need a position list for every query (Section 3.2.1). We are looking at two different approaches to handle the merging of the inverse position lists.

External Merging Either the scan thread just loads an element of the column, probes the predicate index and writes the result to memory. The merging of all position lists is then done after the evaluation of all predicates and not
during a scan. Therefore, the scanning process can be kept simple.

**Internal Merging**  Or the scan thread does not only load an element of the column, but it first loads the inverse position list (bitmap) for that position. Then afterwards it only loads the next attribute value of the column if not the whole bitmap is zero (for further improvements, see Section 3.2.1). The probing of the predicate index may return a bitmap and so after the probing, the scan thread then merges the position list itself by AND-ing the two bitmaps.

### 3.3.3 Materialization

The analysis when to use which materialization strategy for column stores, done by Abadi et al. [6] only handles the evaluation of single queries and can’t be used here. Instead we define our own strategies of early and late materializations and have to compare them to each other.

**Late Materialization**  Here the idea is to first scan all columns appearing in a predicate of a query, but not doing any projections at all. Only in a second step, after all the predicates are evaluated, all the projections are done. Therefore we have to scan over some columns twice.

**Early Materialization**  In contrast to late materialization we go over all columns only once, storing the values in addition to the positions in a separate space if there are predicates and projections on the same column. Then in the second scan we do not have to scan the whole column again, but it is enough to scan the previously pre-selected data.

I have also looked at an alternative version of early materialization, where the pre-selection is in the form `<position, value, query-ids>` or `<position, query-id, value(s)>`. With the form `<position, query-id, value>` we can append the values
of further columns to the relevant tuples. This version is closer to the original definition of early materialization. So with this approach we have to scan fewer values (every column is scanned exactly once). However, we have to update the pre-selected output tuples whenever the position list changes, i.e. some of the tuples are discarded. To do so, this structure has to be re-read for every column that appears in a query predicate. But as this structure becomes very complicated and is really huge in the beginning, it always lost due to access latency. Therefore we no longer track this version and stick to the first approach, when we talk about early materialization.

### 3.3.4 Scanning Order

A simple implementation of a column store clock scan would scan through a whole column, one by one, looking at every item (Figure 3.7 left). However we could as well implement a version where we further partition the data horizontally into blocks and then scan through the data in a zigzag way (Figure 3.7 right). This second option has the advantage that we do not need to scan the whole column, but only the blocks that contain items that still can be part of the result of a query.

We again illustrate this with help of an example (see Figure 3.8). Query q1 has a predicate on column A and B and query q2 has a predicate on column A and C. If we would use a simple scan, we had to scan every column, since in column A was a position, that matches query q1 as well as a position, that matches q2. However, with zigzag scan, after scanning the upper block of column A and B, we don’t have to scan the upper block of column C, because only q2 is interested in column C, but no position matched q2 in column A. Since we are only looking at conjunctions of predicates, the upper block of C can be skipped. Similar in the lower blocks column B does not have to be scanned, because no position matched q1 in column A.

A further advantage of the zigzag implementation is that we need only $1/b$ as much space in memory for the position list as without zigzag, where $b$ denotes the number of block partitions. So zigzag scan like compression of the inverse
Figure 3.8: Example Zigzag Scan

position list is also a means to have the size in main memory under control.

To profit even more of the scenario in the example, we can allow clock scan to do jumps on the column, i.e. an item is not read, when we see in the position list, that it certainly is not part in any query result tuple.
Chapter 4

Cost Models

In this chapter we are looking at the tasks done by a single thread responsible for a horizontal data fragment. Since Crescando is CPU bound, but there are tasks in column store clock scan that certainly are access time bound (like the loading of the huge position list), I developed models looking at the CPU-time as well as models considering memory access-time. The total time for a given scenario is then the sum of the CPU-time for that scenario and the access-time for that scenario.

Costs, that are done by the scheduler before the scans and that don’t vary in the different setups we look at, are not part of the models because they would only complicate the model, but not help with finding the best solution. Such costs include the partitioning of the data among the different scan threads or building the query indexes.

In the following we denote the columns appearing in query predicates as \textit{predicate columns}, the columns appearing in query projections as \textit{projection columns} and the columns appearing in some query predicates as well as some projections as \textit{mixed columns}. When using the terms \textit{pure predicate columns} and \textit{pure projection columns} we refer to columns only appearing in query predicates or projections, respectively, but not in both. We define the set of predicate columns as $PRED$ with $|PRED| = n$, the set of projection columns as $PROJ$ with $|PROJ| = m$ and the set of mixed columns as $MIX = PRED \cap PROJ$.

4.1 Assumptions

In the following we are only looking at read only queries that are conjunctions of equality predicates. We also assume to have enough main memory space available for every setup, i.e. we don’t look at memory size limitations. The number of block partitions in a zigzag scan is denoted with $b$. For simplicity we assume that every column $C$ has a selectivity $sel(C)$ and that
every predicate on \( C \) selects \( \text{sel}(C) \cdot |C| \) items of \( C \). Since we have no information about the correlation of the values of the columns we are looking at, we assume them all to be independent of each other and use the expectation to calculate the number of values returned by a query \( q \), i.e. if \( q \) has a predicate on \( C_1, \ldots, C_n \) we expect \( |C| \cdot \prod_{i=1}^{n} \text{sel}(C_i) \) items to be returned by \( q \) (\( \forall i : |C_i| = |C| \)). We also assume the queries to be independent of each other, i.e. if we have two queries \( q_1 \) and \( q_2 \) that return \( s_1 \cdot |C| \) and \( s_2 \cdot |C| \) items of column \( C \), we expect them to return \( (1 - (1 - s_1) \cdot (1 - s_2)) \cdot |C| \) items together (see Explanation). Although these assumptions are not consistent with reality, they are necessary since we don’t have the real dependencies of the queries and the columns and even if we would have them, it would extremely complicate the models.

**Explanation** If a query \( q_i \) returns \( s_i \cdot |C| \) values, the probability that an arbitrary value is returned by \( q_i \) is \( s_i \). Now consider two independent events \( E_1 \) and \( E_2 \) with the probability that event \( E_i \) occurs is \( s_i \). \( E_i \) is the event that an item is in subset \( S_i \subset C \). The subset \( S_i \) can be seen as the set of items query \( q_i \) returns. We therefore want to compute the probability of the compound event \( E_1 \cup E_2 \). This is \( P(E_1) + P(E_2) - P(E_1 \cap E_2) \) [3]. Since \( E_1 \) and \( E_2 \) are independent \( P(E_1 \cap E_2) = s_1 \cdot s_2 \) and hence \( P(E_1 \cup E_2) = s_1 + s_2 - s_1 \cdot s_2 = 1 - (1 - s_1 - s_2 + s_1 \cdot s_2) = 1 - (1 - s_1) \cdot (1 - s_2) \).

### 4.2 Introduction of Different Model Setups

In this section we briefly introduce the algorithms, for which a CPU-time and an access-time model are presented in the next sections.

#### 4.2.1 External Merging - Late Materialization

The idea is to first scan through all predicate columns, not loading the position list. So each item is probed against the query index and then a bitmap vector is produced. After the evaluation of the last predicate, all position lists are merged.

In a second step we scan through all projection columns. The merged position list tells us, whether we need to load an item or not.

#### 4.2.2 Internal Merging - Late Materialization

First we scan through all predicate columns. The position list is loaded for every column. An item of the predicate column is only loaded and probed against the query index if it isn’t disqualified by the position list, yet. After the probing of the position list and the establishment of the bitmap vector, latter is merged with the position list bitmap.

In a second step we scan through all projection columns. The merged position list tells us, whether we need to load an item or not.
4.2.3 External Merging - Early Materialization

In a first step we scan through all predicate columns. Whenever the column also appears in a projection, we load the position list and copy relevant values to an extra space in memory. So later for the projections we don’t have to scan through the whole columns, but only through the pre-selected values. Since we already load the position list for the mixed columns we don’t have to load every item to probe against the query index in that case. After the evaluation of the last predicate, all position lists are merged.

In a second step we scan through the pre-selected values and the pure projection columns.

4.2.4 Internal Merging - Early Materialization

In a first step we scan through all predicate columns. The position list is loaded for every column. An item of the predicate column is only loaded and probed against the query index if it isn’t disqualified by the position list, yet. After the probing of the position list and the establishment of the bitmap vector, latter is merged with the position list bitmap. For every mixed column we copy relevant values to an extra space in memory.

In a second step we scan through the pre-selected values and the pure projection columns.

4.3 CPU-Time Model

In this model we don’t look at the memory accesses at all, but only at the pure time spent in CPU.

4.3.1 External Merging - Late Materialization

In the first step going over all the predicate columns we need to do an index probing and a subsequent establishment of a position list vector for every item in every column. Assuming a column has \(|C|\) items, the cost for every predicate column is \(c_1 + sel(C) \cdot c_2 \cdot |C|\) with \(c_1\) the cost for probing the query index and \(c_2\) the cost for producing the position list bitmap. Because initializing the bitmap with all zeroes is really fast, but writing the ones at the correct places takes time, the cost is dependent on the number of predicates that are fulfilled and hence in our model the selectivity of the column. After the evaluation of the predicate columns, all \(n\) position lists are merged: \(n \cdot c_3 \cdot |C|\) with \(c_3\) being the cost for merging two position list vectors.

In the second step we go over all projection columns and produce the output tuples. For the cost model we want to estimate the number of output tuples of the form \((qid, value)\) that are generated, \(qid\) being the id of a query and \(value\) a valid projection value of that query. The cost for every projection column \(C_i\) is then \(c_4 \cdot \text{numberOfOutputs}(C_i, PRED)\) with \(c_4\) being the cost for appending a value to an output tuple (see Algorithm 4.1).
Algorithm 4.1 numberOfOutputs

\[\{\text{Input: (ActualColumn } c, \text{AlreadyScannedColumns ASC)}\}\]

\[sel \leftarrow 0\]

\textbf{for all query } q \textbf{ do}

\[qsel \leftarrow 0\]

\textbf{if } c \in q.projectionColumns \textbf{ then}

\[qsel \leftarrow \prod_{C_i \in (ASC \cap q.predicateColumns)} sel(C_i)\]

\textbf{end if}

\[sel \leftarrow sel + qsel\]

\textbf{return } sel \cdot |C|

end for

These formulas do not change whether we allow jumps on clock scan or not. The total CPU cost for column store Clock Scan with external merging and late materialization can be seen in formula 4.1.

\[
\sum_{C_i \in PRED} (c_1 + sel(C_i) \cdot c_2) \cdot |C_i| + n \cdot c_3 \cdot |C| + \sum_{C_i \in PROJ} numberOfOutputs(C_i, PRED) \cdot c_4 \cdot |C_i| \tag{4.1}
\]

4.3.2 Internal Merging - Late Materialization

As above we have again the costs \(c_1\) and \(c_2\) for every item we are looking at. However in addition we also do the merging \(c_3\) internally and hence we are not looking at every item in the column anymore. We first assume that there are jumps allowed in Clock Scan. Therefore the cost for every predicate column is \((c_1 + sel(C_a) \cdot c_2 + c_3) \cdot s_a \cdot |C_a|\). \(s_a\) is the percentage of the expected number of items of the actual column \(C_a\) we have to look at and can be computed with help of Algorithm 4.2. The idea is that if we have \(i\) queries accessing a column with selectivity \(sel\), the number or columns returned is \(1 - (1 - sel)^i\) (see Section 4.1).

Algorithm 4.2 Computing \(s\)

1: \{\text{Input: (ActualColumn } c, \text{AlreadyScannedColumns ASC)}\}
2: \[sel \leftarrow 1\]
3: \textbf{for all query } q \textbf{ do}
4: \[qsel \leftarrow 1\]
5: \textbf{if } c \in q.predicateColumns \textbf{ then}
6: \[qsel \leftarrow \prod_{C_i \in (ASC \cap q.predicateColumns)} sel(C_i)\]
7: \textbf{end if}
8: \[sel \leftarrow sel \cdot (1 - qsel)\]
9: \textbf{return } 1 - sel
10: \textbf{end for}

If we do not allow jumps and have \(b \geq 1\) blocks the cost per predicate column is \((c_1 + sel(C) \cdot c_2 + c_3) \cdot t \cdot |C|\) where \(t\) denotes the probability that a whole
block needs to be scanned. An item is part of a query with probability \( s_a \) and does not appear in a query with probability \( 1 - s_a \). We do not have to scan a block if none of the items in the block appears in any query. This happens with probability \( (1 - s_a)^{|C|/b} \) and with probability \( t_a = 1 - (1 - s_a)^{|C|/b} \) we need to scan the whole block. We have to scan \( t_a \cdot b \) blocks of the size \(|C|/b\).

We define \( e_a \) as follows:

\[
e_a = \begin{cases} 
s_a & \text{if jumps are allowed in clock scan} \\
1 - (1 - s_a)^{|C|/b} & \text{if jumps are not allowed and we have } b \geq 1 \text{ blocks.}
\end{cases}
\]

The cost for every projection column stays the same as with external merging. The total CPU cost for column store clock scan with internal merging and late materialization is shown in Formula 4.2.

\[
\sum_{i|C_i \in PRED} (c_1 + sel(C_i) \cdot c_2 + c_3) \cdot e_i \cdot |C_i| \\
+ \sum_{C_i \in PROJ, numberOfOutputs(C_i, PRED)} numberOfOutputs(C_i, PRED) \cdot c_4 \cdot |C_i|
\]

(4.2)

### 4.3.3 External Merging - Early Materialization

In early materialization we do possible projections already in the first scan through the predicate columns. Therefore we need the position list for the mixed columns, but not for the pure predicate columns and thanks to the position list we don’t need to test every item in the mixed columns. The predicate part becomes

\[
\sum_{C_i \in PRED} (c_1 + sel(C_i) \cdot c_2) \cdot |C_i| + \sum_{C_i \in MIX} (c_1 + sel(C_i) \cdot c_2) \cdot e_i \cdot |C_i| + n \cdot c_3 \cdot |C|
\]

In the mixed columns we first copy every item that matches the predicates evaluated so far to a separate space: \( c_5 \cdot \sum_{C_i \in MIX} s_1' \cdot |C| \) with \( c_5 \) the cost for copying a memory item. \( s_1' \) is computed the same way as \( s_1 \), but line 5 in Algorithm 4.2 becomes: if \( c \in q.projectionColumns \) then. The number of the output tuples generated stays the same, no matter how we process it first. Formula 4.3 shows the CPU cost for these dimensions.

\[
\sum_{C_i \in PRED \setminus MIX} (c_1 + sel(C_i) \cdot c_2) \cdot |C_i| + \sum_{i|C_i \in MIX} (c_1 + sel(C_i) \cdot c_2) \cdot e_i \cdot |C_i| \\
+ n \cdot c_3 \cdot |C| + c_5 \cdot \sum_{i|C_i \in MIX} s_1' \cdot |C_i| \\
+ \sum_{C_i \in PROJ, numberOfOutputs(C_i, PRED)} numberOfOutputs(C_i, PRED) \cdot c_4 \cdot |C_i|
\]

(4.3)
4.3.4 Internal Merging - Early Materialization

The predicate part is the same as with internal merging and late materialization (Formula 4.2) together with the memory copying from Formula 4.3. Adding the generation of the output tuples results in Formula 4.4.

\[
\sum_{i: C_i \in \text{PRED}} (e_1 + \text{sel}(C_i) \cdot c_2 + c_3) \cdot |C_i| + c_5 \cdot \sum_{i: C_i \in \text{MIX}} s_i' \cdot |C_i| \\
+ \sum_{C_i \in \text{PROJ}} \text{numberOfOutputs}(C_i, \text{PRED}) \cdot c_4 \cdot |C_i|
\]  

(4.4)

4.4 Access-Time Model

Here I define the model regarding the cache misses. For this I use the notation introduced in Section 2.3.1. The access times can then directly be derived from the cache misses (see Equation 2.1). I assume the query index to always fit into the L1-cache to obtain better performance (Section 3.1). In the following parameter \( b' \) is defined as \( b' = b \) if we have zigzag scan with \( b \geq 1 \) blocks and \( b' = 1 \) if we have a simple scan and allow jumps.

4.4.1 External Merging - Late Materialization

In the first step going over all the predicate columns, for each predicate column we need to scan through the whole column and have \( b \) times \(|C|\) random accesses to the query index \( I \) for this column, i.e. \( M(s_{\text{trav}}(C)) + b' \cdot M(r_{\text{acc}}(|C|, I)) \) cache misses. The merging in the end of the predicate scans takes an additional \( n \cdot M(s_{\text{trav}}(L)) \) cache misses.

\[
e'_a = \begin{cases} 
  s'_a & \text{if jumps are allowed in clock scan} \\
  1 - (1 - s'_a)^{|C|/b} & \text{if jumps are not allowed and we have \( b \geq 1 \) blocks.}
\end{cases}
\]

Then in the second step going over all the predicates we need to go over \( e'_i \cdot |C_i| \) items for each predicate column \( C_i \) as well as over the whole positionList, i.e. \( e'_i \cdot M(s_{\text{trav}}(C_i)) + M(s_{\text{trav}}(L)) \). The parameter \( e'_i \) is defined similar to \( e_i \), but depends on \( s'_i \) instead of \( s_i \). So for the whole clock scan we get the cost in Formula 4.5.

\[
\sum_{C_i \in \text{PRED}} (M(s_{\text{trav}}(C_i)) + b' \cdot M(r_{\text{acc}}(|C_i|, I_i)) + n \cdot M(s_{\text{trav}}(L)) \\
+ \sum_{C_j \in \text{PROJ}} (e'_j \cdot M(s_{\text{trav}}(C_j)) + M(s_{\text{trav}}(L)))
\]  

(4.5)
4.4.2 Internal Merging - Late Materialization

Because we do the merging of the position list internally, for each predicate column we read the necessary items of the column, have for every item looked at an access to the query index and have a cache miss for reading the position list. Therefore a predicate column has

\[ e \cdot M(s_{trav}(C)) + b \cdot M(r_{acc}(e \cdot |C|, I)) + M(s_{trav}(L)) \]

cache misses.

In the second step we then have for each item in every position list an access to the position list and if necessary to the column and hence

\[ e \cdot M(s_{trav}(C)) + M(s_{trav}(L)) \]

cache misses. In Formula 4.6 the number of total cache misses for these dimensions can be seen.

\[ \sum_{C_i \in \text{PRED}} (e_i \cdot M(s_{trav}(C_i)) + b' \cdot M(r_{acc}(e_i \cdot |C_i|, I_i)) + M(s_{trav}(L))) \]

\[ + \sum_{C_j \in \text{PROJ}} (e_j' \cdot M(s_{trav}(C_j)) + M(s_{trav}(L))) \]

(4.6)

4.4.3 External Merging - Early Materialization

With external merging and early materialization, in a first step we go through all predicate columns. Whenever there are also projections on a column, we have to load the position list to pre-select the early values. If we load the position list for a column anyway, we don't have to load every item of that column, but only the items necessary for the pre-selection and the ones that still have to be tested on the query index of this column. For this we introduce the parameters \( s'' \) and \( e'' \). \( s'' \) can be computed using Algorithm 4.2 with line 5 replaced by:

\[
\text{if } c \in (q\text{.predicateColumns} \lor q\text{.projectionColumns}) \text{ then and} \]

\[
e'' = \begin{cases} 
s'' & \text{if jumps are allowed in clock scan} \\
1 - (1 - s'')^{C_i}/b & \text{if jumps are not allowed and we have } b \geq 1 \text{ blocks.} 
\end{cases}
\]

In a second step we go over all pure projection columns as well as over the pre-selected shorter columns and for each of these columns the position list has to be loaded again. The number of cache misses for this scenario can be seen in Formula 4.7.

\[ \sum_{C_i \in \text{PRED}\setminus\text{PROJ}} M(s_{trav}(C_i)) + \sum_{C_i \in \text{MIX}} e''_i \cdot M(s_{trav}(C_i)) + n \cdot M(s_{trav}(L)) \]

\[ + \sum_{C_i \in \text{PRED}} b' \cdot M(r_{acc}(e_i \cdot |C_i|, I_i)) + \sum_{C_j \in \text{PROJ}\setminus\text{PRED}} e_j' \cdot M(s_{trav}(C_j)) \]

\[ + \sum_{C_i \in \text{PROJ}} M(s_{trav}(L)) + \sum_{k=1}^{[\text{MIX}]} (e_k' \cdot M(s_{trav}(O_k)) + M(s_{trav}(L))) \]

(4.7)
4.4.4 Internal Merging - Early Materialization

Here again, we go through all predicate columns, in a first step. Since we do the merging internally we have to load the position list for every column and have a cache misses for the items of every column.

The second step is the same as with external merging - early materialization. Formula 4.8 shows how to compute the cache misses for column store Clock Scan using early materialization with internal merging.

\[
\begin{align*}
\sum_{C_i \in \{\text{PRED} \setminus \text{PROJ}\}} e_i \cdot M(s_{\text{trav}}(C_i)) &+ \sum_{C_i \in \text{MIX}} e''_i \cdot M(s_{\text{trav}}(C_i)) \\
&+ \sum_{C_i \in \text{PRED}} (b' \cdot M(r_{\text{acc}}(e_i \cdot |C_i|, I_i)) + M(s_{\text{trav}}(L))) \\
&+ \sum_{C_j \in \{\text{PROJ} \setminus \text{PRED}\}} (e'_j \cdot M(s_{\text{trav}}(C_j)) + M(s_{\text{trav}}(L))) \\
&+ \sum_{k=1}^{\text{|MIX|}} (e'_k \cdot M(s_{\text{trav}}(O_k)) + M(s_{\text{trav}}(L)))
\end{align*}
\]

(4.8)
Chapter 5

Evaluation

In this chapter we want to evaluate the models introduced in Chapter 4.

5.1 Amadeus Workload

Like Crescando, we look at the Amadeus workload. Since the data of Amadeus is confidential we had no access to the actual data. However we were provided the statistics of a dataset of 8 million records, that was also used in the Crescando paper [14]. With help of this statistics we generated around 4000 queries to feed in our model to compare the different setups and find out, which strategy works best.

Amadeus stores all flight bookings in a single table, consisting of 48 attributes. Figure 5.1 shows the probabilities of the projections to be part in a query. Most queries have at least 27 projections.

By analyzing the predicate distribution we first see, that 56% of the queries target the eight predicate attributes PROVIDER, PRODUCT_ID, DATE_IN_FIRST_LEG, DATE_IN, SGT_QUALIFIER, CABIN, BOOKING_STATUS and CANCEL_FLAG together, in fact 99.5% of all queries have a predicate on PROVIDER, PRODUCT_ID, DATE_IN and CANCEL_FLAG. The second most frequent form of queries with 6.1% target the nine selection predicates PROVIDER, PRODUCT_ID, ALPHA_SUFFIX, DATE_IN, CTY_FROM, CTY_TO, RLOC, SGT_QUALIFIER and CANCEL_FLAG. In Figure 5.2 we see the probabilities of the attributes to occur in a selection predicate.

On the basis of the different values and its occurrences in the dataset, we calculated the selectivities of the different columns. In Figure 5.3 the selectivities of the columns occurring in a predicate of our evaluation setup can be seen. Because we have no statistics on the distribution of RLOC, we pessimistically assume it to be 1, i.e. that there is no selectivity on this attribute.
Figure 5.1: Projection Attribute Occurrences

Figure 5.2: Predicate Attribute Occurrences

Figure 5.3: Predicate Selectivities
### Table 5.1: Time Assumptions

<table>
<thead>
<tr>
<th>( \lambda_{L2} )</th>
<th>5ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda_{Mem} )</td>
<td>30ns</td>
</tr>
<tr>
<td>( c_1 )</td>
<td>10ns</td>
</tr>
<tr>
<td>( c_2 )</td>
<td>10ns</td>
</tr>
<tr>
<td>( c_3 )</td>
<td>10ns</td>
</tr>
<tr>
<td>( c_4 )</td>
<td>5ns</td>
</tr>
<tr>
<td>( c_5 )</td>
<td>1ns</td>
</tr>
</tbody>
</table>

5.2 Comparison of Model Setups

To analyze the different dimensions and to compare the different setups with each other, we assume a cache hierarchy with two levels of cache (however today, there are mostly three cache levels). We expect the L1 cache to have 1024 cache lines of size 64 byte and the L2 cache to have 16384 cache lines of size 512 byte. Further we assume a column to consist of 3 million rows. For simplicity we also expect, that always the whole index is loaded, that it always fits into the L1 cache and that it stays there, until the column is fully scanned. Because we are looking at equality predicates, we have hash indexes and hence assume the size of an index to be the sum of the size of all query ids with a predicate in the index plus the size for all different values appearing in the predicates. For our evaluation we assume the times displayed in Table 5.1.

We first compare the different setups of Chapter 4 assuming we have uncompressed position lists. Then later we look at the impact of compressing the position lists.

#### 5.2.1 Uncompressed Position List

**Analysis of Column Order**

We want to analyze the impact of the order the columns are scanned through, depending on the selectivities of the columns. Therefore we look at a version where we scan the columns sorted by their selectivities in descending order (highly selective columns last), a version with random order and a version where the columns are sorted by their selectivities in ascending order (highly selective columns first). Additionally we also look at a version where the columns are sorted by the product of their selectivity and their width in ascending order. The most selective column has selectivity 0.001669 and the least selective column has selectivity 1. However, we are not interested in a comparison between the different merging and materialization methods here. For this experiment we allow clock scan to do jumps. The result of this experiment can be seen in Table 5.2.

The differences between the different setups are not that big. This is because most of the time is lost in the access of the huge position lists which has to be loaded the same number of times for a specific model, no matter what scanning
order we are using. Further the differences between the last two scanning orders is only in the 4th decimal place and the better version probably is dependent on the workload. In our example with the Amadeus workload, the difference in the order, the columns are scanned, is rather small too.

In the following of this section we are always looking at the columns sorted by their selectivities in ascending order unless specified otherwise.

<table>
<thead>
<tr>
<th>Analysis of Scanning Order</th>
</tr>
</thead>
</table>

Here we want to analyze the difference between allowing jumps in clock scan and not allowing them. In the latter case we also look at how zigzag scan improves performance, see Table 5.3.

As expected, allowing jumps results in the fastest evaluation time. When we don’t allow jumps it is better to do zigzag scan. However there is a maximal number of block partitions that should not be exceeded since the index has to be loaded more often because an index falls out of the cache as all the indexes for the other predicates are loaded. The best number of block partitions for the zigzag scan strongly varies between the different merging and materialization strategies used and has to be found experimentally for a specific workload.

In the following of this section we always assume jumps to be allowed unless specified otherwise.
## Evaluation - Comparison of Model Setups

<table>
<thead>
<tr>
<th></th>
<th>external merging, late materialization</th>
<th>internal merging, late materialization</th>
<th>external merging, early materialization</th>
<th>internal merging, early materialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jumps allowed</td>
<td>14.26s</td>
<td>13.31s</td>
<td>16.96s</td>
<td>13.35s</td>
</tr>
<tr>
<td>Jumps not allowed (no zigzag)</td>
<td>14.35s</td>
<td>14.35s</td>
<td>17.61s</td>
<td>14.37s</td>
</tr>
<tr>
<td>Jumps not allowed (100 blocks)</td>
<td>14.33s</td>
<td>14.15s</td>
<td>17.49s</td>
<td>14.17s</td>
</tr>
<tr>
<td>Jumps not allowed (500 blocks)</td>
<td>14.34s</td>
<td>13.95s</td>
<td>17.37s</td>
<td>13.98s</td>
</tr>
<tr>
<td>Jumps not allowed (1000 blocks)</td>
<td>14.36s</td>
<td>13.85s</td>
<td>17.31s</td>
<td>13.88s</td>
</tr>
<tr>
<td>Jumps not allowed (2500 blocks)</td>
<td>14.40s</td>
<td>13.76s</td>
<td>17.27s</td>
<td>13.79s</td>
</tr>
<tr>
<td>Jumps not allowed (5000 blocks)</td>
<td>14.48s</td>
<td>13.76s</td>
<td>17.31s</td>
<td>13.79s</td>
</tr>
</tbody>
</table>

Table 5.3: Jumps vs No Jumps vs Zigzag
Table 5.4: External vs Internal Merging

<table>
<thead>
<tr>
<th></th>
<th>late materialization</th>
<th>early materialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>External Merging</td>
<td>14.26s</td>
<td>16.96s</td>
</tr>
<tr>
<td>Internal Merging</td>
<td>13.32s</td>
<td>13.35s</td>
</tr>
</tbody>
</table>

Table 5.5: Late vs Early Materialization

<table>
<thead>
<tr>
<th></th>
<th>external merging</th>
<th>internal merging</th>
</tr>
</thead>
<tbody>
<tr>
<td>Late Materialization</td>
<td>14.26s</td>
<td>13.32s</td>
</tr>
<tr>
<td>Early Materialization</td>
<td>16.96s</td>
<td>13.35s</td>
</tr>
</tbody>
</table>

**Analysis of Merging Strategies**

The two merging strategies are compared in Table 5.4. An implementation with internal merging seems to be better than one with external merging. That makes sense, because if we do the merging internally, we profit from the position list and hence do not have as much data items to load and probe against the query index as without knowing anything of the previous predicate evaluations. In early materialization the difference is even bigger since we often load the position list already when scanning through a predicate column (in every mixed column).

**Analysis of Materialization Strategies**

The comparison between early and late materialization can be seen in Table 5.5. Late materialization wins over early materialization. This is clear as since we allow Clock Scan to do jumps, we don’t have to scan the whole column anyway and we don’t profit from pre-selecting data.

More interesting is to analyze what materialization strategy is better if we do not allow Clock Scan to do jumps and hence a whole column has to be scanned (when for example jumps are too hard to implement or we don’t profit enough because the compiler or the CPU itself does that much prefetching that the advantages of jumps diminish). However as can be seen in Table 5.3 by looking at the different numbers in a row, even without doing jumps late materialization is faster. So the overhead of copying values does not pay off.

**Conclusion**

So without any compression on the position list, the best solution is an algorithm, that scans the columns sorted by their selectivities in ascending order, that allows Clock Scan to do jumps on the columns and that uses late materialization while instantly merging the position lists internally. The times don’t differ much though, because they all have the loading of the big position list as bottleneck.
### 5.2.2 Compressed Position List

Since loading the position list is the most time consuming part, we want to see, how compression of the position list affects the result. We assume that the merging is done directly on the compressed position list, which is possible if the correct compression algorithm is used. We just look at how a compression to 10% or 1% of its original size affects the results. The real rate of compression strongly depends on the selectivity, though. As seen in Table 5.6, compression leads to much faster results: Clock Scan is no longer access-time bound. This also effects the comparisons we have done in the previous section. The only thing that changes is that early materialization becomes the better materialization strategy at external merging. However since internal merging stays clearly faster, we don’t further pursue external merging.

Playing around with the time values could not change the winning strategy, only the losing. No matter which value is increased or decreased the fastest approach always stays the following:

- Sort predicate column according to their selectivities in ascending order
- Allow Clock Scan to do jumps
- Use internal merging
- Use late materialization

Until now we have always assumed the total time to be the sum of the access time and the CPU time. However in reality these two times overlap and we could as well look at the total time being the maximum of the two in the other extreme. This doesn’t change the winning strategy either, though. It actually has the same effect as massively increasing, respectively decreasing the L2 and

<table>
<thead>
<tr>
<th>Position List</th>
<th>External Merging, Late Materialization</th>
<th>Internal Merging, Late Materialization</th>
<th>External Merging, Early Materialization</th>
<th>Internal Merging, Early Materialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncompressed Position List</td>
<td>14.26s</td>
<td>13.31s</td>
<td>16.96s</td>
<td>13.35s</td>
</tr>
<tr>
<td>Compressed to 10% of Original Size</td>
<td>2.75s</td>
<td>1.80s</td>
<td>2.52s</td>
<td>1.84s</td>
</tr>
<tr>
<td>Compressed to 1% of Original Size</td>
<td>1.60s</td>
<td>0.65s</td>
<td>1.08s</td>
<td>0.68s</td>
</tr>
</tbody>
</table>

Table 5.6: Compression of Position List
As it is not clear if allowing jumps in Clock Scan is realizable since we can’t predict the CPU’s behavior like prefetching, we want to investigate the best number of block partitions in a zigzag approach for the winning strategy (predicate columns sorted ascending to their selectivities, internal merging and late materialization). The results are shown in Figure 5.4.

Figure 5.4: Best Number of Block Partitions of Zigzag Scan for Internal Merging, Late Materialization, Position List Compressed to 10%
Chapter 6

Conclusion and Future Work

6.1 Conclusions

There are many different approaches of how we can implement Clock Scan on a column store. This thesis analyzed different approaches and their advantages and disadvantages. We looked at the main differences in merging of the position lists between a scenario where we look at one query at a time and a Clock Scan where we execute thousands of queries together. In this context we introduced the concept of the inverse position list. Then we analyzed two different data partitions (horizontal and vertical) among the scan threads and its influence of a column store Clock Scan realization. We restricted the more accurate analysis on a horizontal data partitioning.

We thought of different dimensions that may impact the running time of a column store Clock Scan and then showed with help of an analytical cost model which combination of strategies leads to the fastest result. The winning dimensions are to scan the columns sorted by their selectivities in ascending order, to do the merging of the position lists internally, right after the probing of a query index, to do late materialization and to allow Clock Scan to do jumps on a column. However these models had to rely on many assumptions regarding the cache hierarchies and replacement strategies as well as independency of the different columns and queries. These assumptions sometimes certainly contradict reality. A CPU may also do cunning prefetching and other optimizations that we are not aware of and that are not contained in the cost models. Therefore a real implementation of the different strategies may lead to other results. However the ideas needed are the same.

Aside from said issues an implementation faces additional problems like space limitations in main memory, since the position lists get really huge. Therefore zigzag scan comes in handy because the position list that is in memory at a time is smaller. Compression of the position list may also help.

It’s not clear, either, if the column store implementation gets faster than the row store implementation, because the handling of the position lists may evict
the advantages gained by scanning less data. Unfortunately, time ran short in the end. So I preferred to neatly finish the theoretical analysis instead of doing a fast, short implementation. A next time I probably would do the implementation parallel to the development of the analytical cost model, to see which additional problems arise in reality and to include them in the cost models.

6.2 Future Work

There is still a lot of future work that can be done. First of all, a simple version of column store Clock Scan is implemented. This version restricts itself to the same assumptions as the models in this thesis: an implementation of a single thread, that handles the evaluation of many queries on a single table, and a read only workload with conjunction of equality predicates. At the moment of completion of this thesis, T. Salomie, a PhD student is working on an implementation of column store Clock Scan. Then it has to be compared with the existing row store implementation of [14]. If the column store implementation proofs to be at least as fast as the row store implementation, the implementation can be completed by adding the possibility to handle range predicates and null values. It also has to be analyzed how to build the predicate indexes as fast as possible. The best solution may not be a pure column store, but contain parallel n-column Clock Scans if two predicates very often appear together, for instance in a compound primary key.

In a further step the handling of UPDATE and DELETE statements can be added. There the main problem will be to guarantee data consistency. With a horizontal data partitioning we can profit from the work of Crescando where a solution was presented.

We also could consider to expand column store Clock Scan to handle multiple tables. For this the considerations to a vertical partitioning can come in handy.
Bibliography


