Master Thesis

Multi-query optimization in a scan-based relational main-memory table

Author(s):
Steiner, Ralf

Publication Date:
2010

Permanent Link:
https://doi.org/10.3929/ethz-a-006207000

Rights / License:
In Copyright - Non-Commercial Use Permitted
Multi-query Optimization in a Scan-based Relational Main-memory Table

Master Thesis
Ralph Steiner

Supervised by:
Prof. Donald Kossmann
Philipp Unterbrunner

Systems Group, http://systems.ethz.ch/
Department of Computer Science
ETH Zurich

May, 2010 - October, 2010
Abstract

In database systems the term multi-query optimization traditionally refers to the practice of finding common sub-expressions between multiple queries, and sharing intermediate results between them. However, in a scan-based relational main-memory table the task of multi-query optimization is a completely different one. There is just one table, which is continuously scanned and while scanning, a large set of queries is concurrently executed on each record. In such a system, queries are typically small, and sharing intermediate results is difficult. Also it has no indexes on the data. Instead, it builds predicate indexes on the set of active queries. Here, multi-query optimization refers to the practice of finding a set of indexes, such that the cost of concurrently executing these queries is minimized.

Such an optimization consists of two parts, estimating the cost of a set of indexes and enumerating various possible sets of indexes. With a precise cost estimation, one can accurately pick the fastest set from the enumerated index sets. Cost estimation requires statistical data about the data in the table. The thesis shows how one can gather and provide statistical data in a scan-based system. For this purpose, data structures and algorithms known from streaming applications are employed. Furthermore, the thesis presents an elaborated cost estimation model based on CPU costs of different index types. Finally, this thesis shows that the optimization problem itself is NP-hard. However, it presents approximation algorithms to find some good index sets.
Acknowledgments

I would like to express my gratitude to my supervisors Prof. Dr. Donald Kossmann and Philipp Unterbrunner for their help and guidance through the course of this work. I would also like to thank all my friends who supported me during this thesis.
Contents

1 Introduction .............................................. 1
   1.1 Motivation & Problem Statement .................. 1
   1.2 Contribution ........................................ 2
   1.3 Outline & Approach ................................ 2

2 The Crescando Storage System .......................... 5
   2.1 Overview .......................................... 5
   2.2 Clock Scan ......................................... 6
   2.3 Index Union Join Plan .............................. 7
   2.4 Data Structures .................................... 7
   2.5 Statistics Manager ................................. 8
   2.6 Baseline Optimizer ................................. 9

3 Selectivity Estimation ................................... 11
   3.1 Constraints & Design .............................. 11
   3.2 Related Work ....................................... 12
   3.3 Implementation .................................... 12
      3.3.1 The CountMin Sketch ......................... 13
      3.3.2 The FastAGMS Sketch ......................... 13
      3.3.3 Optimizations ................................ 14
   3.4 Evaluation ......................................... 14
      3.4.1 Setup .......................................... 14
      3.4.2 Depth Parameter ............................... 15
      3.4.3 Accuracy ...................................... 17
      3.4.4 Speed ......................................... 18

4 Cost Model .................................................. 19
   4.1 Design ............................................. 19
   4.2 The Baseline Cost Model .......................... 20
      4.2.1 GreedyOptimizer ............................... 21
   4.3 The Index-aware Cost Model ....................... 22
      4.3.1 Constant Initialization ....................... 22
   4.4 Related Work ....................................... 23
   4.5 Index Costs ........................................ 23
## CONTENTS

4.5.1 JaggedArray ............................................. 23  
4.5.2 MultiHashMap .......................................... 24  
4.5.3 OneDimRTree ........................................... 25  
4.6 Design Notes ............................................... 27  
4.7 Evaluation .................................................. 27  
 4.7.1 Setup ...................................................... 27  
 4.7.2 Accuracy of JaggedArray ................................ 28  
 4.7.3 Accuracy of MultiHashMap ............................ 29  
 4.7.4 Accuracy of OneDimRTree ............................ 30  
 4.7.5 Accuracy for Multiple Indexes at the Same Time .. 31

5 Plan Enumeration ........................................... 33  
 5.1 The Optimization Problem ............................... 33  
    5.1.1 Exact Cover ........................................... 33  
    5.1.2 Minimum Exact Cover ............................... 34  
    5.1.3 NP-Hardness of Minimum Exact Cover ............. 34  
    5.1.4 The Index Union Join Optimization Problem ...... 35  
    5.1.5 Search Space of the Index Union Join Optimization Problem ........................................... 36  
 5.2 Related Work ................................................. 36  
 5.3 Optimizers .................................................. 37  
    5.3.1 K-Greedy Distribute .................................. 37  
    5.3.2 K-Backtrack Distribute .............................. 39  
    5.3.3 Index Merger .......................................... 40  
 5.4 Evaluation .................................................. 42  
    5.4.1 Index Quantity Test .................................... 42  
    5.4.2 Index Quality Test .................................... 43  
    5.4.3 Discussion ............................................. 44

6 Overall Evaluation ......................................... 45  
 6.1 Setup ....................................................... 45  
    6.1.1 Test Platform .......................................... 45  
    6.1.2 Amadeus Schema & Data Distribution .............. 45  
    6.1.3 Amadeus Workload ...................................... 45  
    6.1.4 Synthetic Workload .................................... 46  
    6.1.5 Manually Corrected Cost Model ..................... 46  
    6.1.6 General Benchmark Settings ....................... 47  
 6.2 Evaluation .................................................. 47  
    6.2.1 Performance of the Novel Selectivity Estimation .... 48  
    6.2.2 Performance of the New Cost Model ................. 49  
    6.2.3 Comparison of the Novel Plan Enumeration Algorithms .. 50  
    6.2.4 Baseline vs. New Optimal Approach ................. 53

7 Conclusion .................................................. 55  
 7.1 Future Work ............................................... 55
Chapter 1

Introduction

1.1 Motivation & Problem Statement

Multi-query optimization in a scan-based relational main-memory table is a novel problem and completely different from multi-query optimization in a traditional database management system (DBMS). A scan-based relational table is continuously scanned and while scanning, a large set of queries is concurrently executed on each record. There are no indexes on the data, but instead, such a system could build indexes on the set of active queries. Here, multi-query optimization is the technique of determining a set of query-indexes such that the queries are executed in the most effective way.

This thesis uses Crescando as an implementation of a scan-based relation main-memory table. Crescando was developed at ETH Zurich as a system that offers predictable performance for unpredictable workloads [29]. It is based on the idea that executing a query on one table through a full scan takes about the same amount of time, no matter what the query looks like. But as Crescando executes multiple queries through the same scan, the scan time depends on the number of queries. To increase the scan performance, Crescando builds indexes on the set of queries sharing a scan cursor.

However, the current optimizer of Crescando not always builds the optimal set of query indexes. This is especially the case when there are more indexes than one or two needed, to index most of those queries. Crescando uses a cost based optimizer and inaccurate cost estimation is one reason why poor plans are built. Another reason is the relatively simple plan enumeration algorithm that is based on a greedy policy.

Cost estimations are in general based on two things: statistical data, and formulas using this data. For the current optimizer, selectivity estimates are of particular interest. Yet, Crescando estimates selectivity uniformly for an attribute: the selectivity of an equality predicate on attribute $A$ is estimated in terms of the number of unique values in $A$, no matter on what value this predicate matches. Consequently, Crescando needs support for selectivity estimation on a per-value level. Unlike in traditional database systems, where histograms are built on data indexes, Cres-
cando has to investigate different ways, because it is a scan-based table and has no indexes on the data.

Besides the statistical data, this thesis will show what the cost estimation formula for such a system could look like. A traditional DBMS often uses a mix of the estimated CPU and I/O costs to model the actual cost of a plan. Because all data resides in main memory, Crescando has no traditional I/O costs. This thesis presents a cost model, where the cost of a query-index is estimated with respect to its type. Instead of I/O costs, the cache footprint of an index would be an interesting cost measure too, but this will not be covered.

As mentioned before, the simple plan enumeration algorithm is another reason for sub-optimal query-index sets. This thesis proves that the optimization problem is NP-hard, and because the search space is enormous, an exhaustive search is not feasible. This thesis then presents some novel approximation algorithms for this problem and evaluates them.

Increasing the performance of an index-aware workload, where each query fits into the same index, seems not to be reasonable. However, as the motivation behind Crescando is a predictable performance on unpredictable workloads, the resulting system should be faster for a diverse and unpredictable workload than the current system.

1.2 Contribution

The main contributions of this thesis are as follows:

- We show how one can collect, under tight resource constraints, statistical data in a scan-based relational main-memory table.

- We present a model to estimate the cost of a scan in a main memory table and show how these costs can be estimated in the presence of different predicate index-types.

- The thesis gives a formal proof of the NP-hardness of the optimization problem and presents multiple approximation algorithms.

1.3 Outline & Approach

Chapter 2 introduces Crescando and presents the parts of it which are the most important for this thesis. Each of the next three chapters presents the work on one optimization area: chapter 3 treats the selectivity estimation, chapter 4 presents the work on the cost model and chapter 5 deals with the optimization problem itself and presents novel approximation algorithms. Whereas we gave in this chapter global aims and sketched the goals of each optimization area, each of the three chapter starts with its own problem statement. Each optimization chapter concludes itself with a small evaluation section. Chapter 6 gives an overall evaluation.
on the whole system. In the final chapter 7, we conclude the work of this thesis and give some ideas on future work.
Chapter 2

The Crescando Storage System

This chapter gives a short overview of the Crescando storage system and provides a deeper look at some parts, which are important for this thesis. For more details on other parts of Crescendo, please refer to paper [29].

2.1 Overview

Crescendo consists mainly of four parts. The main functionality resides in the storage library. The schema-dependent library is linked to this library. The schema library contains only highly performance-relevant code and code that depends on it. It is generated automatically with the use of the Crescando schema compiler that uses a SQL-like table description as input. Currently, a schema can have primitive types, dates and strings. The last part of Crescando is the storage node, which links against a schema-dependent library. In essence, it provides a network interface that can be used for a distributed system. Other client applications may link against a C interface provided by the schema-dependent library.

Crescendo has a row-based storage layout. Rows are aggregated together in so-called chunks. A chunk is comparable to a data page in a traditional DBMS. It also has slots for rows but it is optimized for main memory usage (e.g. its size de-
pends on the cache size). Each chunk belongs to a segment. As Crescando supports multiple threads for scanning, each scan thread has its segment.

Besides the read queries (select), there are three different write queries: update, delete and insert. All the queries except for the insert have a conjunction of predicates. Predicates have the form of \(<attribute, \text{operator, constant}>\), where \text{operator} can be an equality, range or null check operation.

2.2 Clock Scan

Crescando uses a novel scan algorithm, the so-called Clock Scan. Crescando scans the data and executes several queries at the same time. Concurrently arriving new queries are buffered in queues, one queue for read and one for write queries. Each scan does the following steps:

1. Activation phase: First, the algorithm takes write queries out of their queue and adds them to the set of queries that will be executed on the data. It does then the same with the read queries in their own set. ‘Activating’ a query refers to this process.

2. Optimization phase: An optimizer builds for each set a query plan. For the reminder of this thesis, \textit{optimization time} denotes the time that is used for this phase.

3. Execution phase: Now the scan applies the query plans on each chunk of its segment. First, the write-query plan is executed and then, the read-query plan is executed on the same chunk. It can easily be proven that if the write queries are executed strictly in their order of arrival, the read queries will always see a consistent snapshot of the data, when they are executed afterwards. Generated results are placed in a common \textit{result queue}. Corresponding to the optimization time, the time needed for this phase is referred to as the \textit{execution time}.

4. Deactivation phase: After all chunks have been scanned the queries are deactivated. Typically, a query just places an \textit{endofstream} result on the result queue.
2.3 Index Union Join Plan

To execute multiple queries on a chunk in the execution phase of the scan, Crescando is currently using the Index Union Join plan. This plan consists of a set of predicate indexes \( \mathbf{is} \) and a set of unindexed queries \( \mathbf{qs} \). Each query of the plan is either indexed by exactly one index or in the set of the unindexed queries. Figure 2.3 taken from [29] visualizes this plan.

The plan is executed for each record in the current chunk and consists of two parts. In the first part, each query in the unindexed query set \( \mathbf{qs} \) is checked, if its predicates matches with the record. In the context of this thesis, this test is referred to as *query evaluation*. In the second part, each index is probed with the values from the record. Queries that are returned by the probing are then fully evaluated. This indirection is beneficial because an index may contain hundreds of queries but probing returns only a few queries to evaluate. In case of a read-query plan, for each query that matched, a projected record is added to the result queue. In case of the write-query plan, the matched queries are executed in their arrival order to guarantee consistency.

2.4 Data Structures

Crescando currently supports three different types of indexes.

- **JaggedArray**: A jagged array is a two dimensional array, where each row can have a different size.

  The index uses the attribute value as offset to access a row. In the rows are the pointers to the queries, which have a predicate on that value. Figure 2.4 illustrates a JaggedArray for booleans, where six queries have an equality predicate on the value ‘true’. This index is only useful for attributes with a small domain size such as a boolean attribute, because it needs a row for every domain value.
• **MultiHashMap**: This index is a simple hash map that uses linear probing as collision resolution. Linear probing is chosen, because it has good cache locality. If two keys have not only the same hash value but also the same value, the elements are chained together in a linked list. So all queries with the same predicate constant are in the same list. In figure 2.5, the two keys ‘Leif’ and ‘Hagbard’ have a hash collision, and queries ‘Q7’ and ‘Q4’ are chained because they have the same key.

• **OneDimRTree**: For range queries, Crescando uses a one-dimensional R-Tree. The tree consists of leaf nodes containing the intervals of the elements. These nodes are sorted by their interval midpoint. Interior nodes consist of child-node references and an interval. This interval spans the child-node intervals. The number of child-node references is called the fan-out. Crescando uses as default a fan-out of four. The figure 2.6 visualizes the tree for a fan-out of two and its probing. Note that probing may require following several branches.

### 2.5 Statistics Manager

To find a query plan, the optimizer uses selectivity estimates, which are gathered by the statistics manager. The manager provides an operation to refresh all statistics. It is called after bulk loading, and after a certain fraction of the table has been modified. The manager can update the statistics on a per-attribute basis.

Currently, the selectivity estimates are based on a small set of statistics: number of records and for each attribute, the number of null values, as well as the number of distinct values. To gather these estimates, the manager enqueues a special unconditional read query for each attribute. Such a query matches every record, but instead of putting the record on the result queue, a counter is updated. There are two counters maintained, one for null values and a probabilistic counter [9, 30]. When such a query gets deactivated (see section 2.2), it estimates the number of distinct values with the probabilistic counter and informs the statistics manager which in turn updates the statistics.
2.6 Baseline Optimizer

Algorithm 1: Baseline Optimizer

Data: Gain thresh;
Input: OpSet os; AttributeSet A;
Output: IndexSet is ← ∅; QuerySet uos ← os;
repeat
    Attribute a ← maxRangeGain\{a ∈ A\};
    Gain gain ← Gain(a);
    Gain rangeGain ← RangeGain(a);
    if rangeGain > thresh then
        Index idx;
        if gain > thresh then
            idx ← BuildIndex(a, uos);
            if rangeGain = gain > thresh then
                idx ← idx ∪ BuildRangeIndex(a, uos);
            end
        else
            idx ← BuildRangeIndex(a, uos);
        end
        is ← is ∪ idx;
        uos ← uos \ {q ∈ idx};
        A ← A \ {a};
    end
until rangeGain ≤ thresh;

The current optimizer of Crescando uses a greedy algorithm shown in Algorithm 1. At each iteration, it decides to build an index on the attribute with the highest expected range gain. The range gain is a gain computed on all predicates of a query, whereas the normal gain does not consider range predicates. The value thresh is used to prevent the optimizer from building too many indexes. The function BuildIndex builds, depending on the attribute type, a MultiHashMap or a JaggedArray and the BuildRangeIndex builds an OneDimRTree. To build an index, all queries that are covered by this index (i.e. having a predicate on this attribute with an appropriate operation) are inserted in it. Queries that were inserted are taken out of consideration for following iterations. The range gain function is defined as:

\[
\text{RangeGain}(Q, a) := \sum_{q ∈ Q} (1 - \text{selectivity}(a, q))
\]

This function is based on the following heuristic: As mentioned in section 2.3, the more queries can be excluded by indexes, the less have to be evaluated. The probability of evaluating a query \( q \) after probing a predicate index on attribute \( a \) is the selectivity of \( q \) with respect to \( a \). The gain is then simply the sum of the probability of not evaluating a query given an index on \( a \).
Chapter 3

Selectivity Estimation

As already mentioned in section 2.5, Crescando only has a small set of statistics. This chapter shows how one can improve the current selectivity estimation.

If the optimizer has better estimates of how many times a query predicate matches, it can build better plans by choosing the index with the least expected number of evaluations for this query. To do this estimate the optimizer uses statistical data to compute selectivity estimates of predicates.

However, the current optimizer can only do an exact estimate for null values. Selectivity of other values of an attribute is estimated uniformly according to the number of unique values. This is not an issue for more or less uniform distributed values, but realistic workloads, such as the Amadeus\(^{(1)}\) workload, do often have skewed data distributions.

3.1 Constraints & Design

A traditional database management system estimates selectivity often with the help of histograms. These histograms are built primarily on top of data indexes. Because Crescando has no data indexes, statistical data must be gathered in a scan-based fashion. In Crescando, the statistical data can be gathered while executing queries, but in order to not noticeably decrease query execution performance, this should be as fast as possible. Also it is important that the metadata collector uses little space. The more memory is used, the higher the cache footprint of a scan phase in which metadata is gathered. This may lead to additional cache trashing and hence to a lower performance. These considerations can be applied to the use case of requesting metadata also. When the optimizer requests for selectivity estimates, the system should not loose too much performance.

To collect statistical data, a special unconditional read query is issued, similar to what is already done for the current set of metadata (see section 2.5). This Detailed-StatisticsSelect contains the actual metadata collector, the Counter. The Counter has the operation count(in value : Object), which is called for each row. When

\(^{(1)}\)Service provider for managing travel-related bookings. For details see section 6.1.2
the query gets deactivated, the overall metadata (the Selectivities object) is updated with the values from the Counter. The Counter needs two other operations: `pointEstimate(in value : Object) : int`, which estimates the number of occurrences of the given value, and `rangeEstimate(in range : Range) : int`, which does the same, but for a whole range. The optimizer calls them to estimate predicate selectivity.

### 3.2 Related Work

Algorithms and data structures known from data streaming applications seem naturally applicable to our case. The paper [5] gives a good overview of possible Counter implementations. One way is just to count the k most-frequent items and estimate the frequency of other items. Such implementations are called counter-based algorithms in the literature. To this group belong Frequent (also known as MisraGries) [13, 20], LossyCounting[18] and SpaceSaving[19], which promise the highest updated speed and precision in this group.

Alternatively there are quantile algorithms like GK algorithm[11] or QDigest[26], but they do not promise a good update speed.

The last group are called sketches[24]. They support frequency estimation of low frequent items out of the box. Well-known candidates of this group are the AGMS[1], FastAGMS[4], which is a faster refinement of AGMS, FastCount[28] and CountMin[6]. FastCount provides the same error bounds and update speed as FastAGMS, but experiments [24] have shown that FastAGMS is more accurate. CountMin promises a higher accuracy for skewed data than FastAGMS. The authors of [5] state that sketches have a lower update speed than the counter-based algorithms, but they were interested only in the k-most frequent items. They modified the sketches in order to be as space saving as possible, which resulted in higher update cost.

### 3.3 Implementation

Obviously, there is no need for a complicated Counter if the attribute type has a small domain size like booleans or chars. To give a precise estimate for a boolean type, just `true` and `false` values have to be counted (`null` values are already counted). This needs two integer fields. To count attributes having a one-byte size type, an implementation having an array of 256 integers is also good enough. For attribute types of a bigger size, a different solution is required. We have chosen sketches, because they support estimation of low frequent items and promise an update speed of 50-400ns [24]. As argued in the last section, the FastAGMS and CountMin were selected as most promising sketches.
3.3 Implementation

3.3.1 The CountMin Sketch

The CountMin sketch [6], or CM for short, owes its name to the two basic operations of this data structure. At construction time, CM is obviously counting and to get a frequency estimate it computes a minimum. Figure 3.1 illustrates the CountMin sketch. The sketch consists of a two-dimensional array of counts with width $w$ and depth $d$: $\text{count}[0,0]...\text{count}[d-1, w-1]$.

Additionally, it has $d$ hash functions $h : \{0...n\} \rightarrow \{0...w-1\}$, chosen uniformly at random form a pairwise-independent hash family. To count an item, the sketch increases in each row one counter, which is chosen with the use of the hash function associated to this row. Let $i \in \{0...n\}$ denote an item. Formally, counting $i$:

\[ \forall j \in \{0...d-1\} : \text{count}[j, h_j(i)] \leftarrow \text{count}[j, h_j(i)] + 1 \]

Let $a_i$ be the number of counts of the item $i$ and $\hat{a}_i$ the estimated counts. To estimate an item frequency, the minimum count value is taken from the counters of this item:

\[ \hat{a}_i = \min_{j \in \{0...d-1\}} \left( \text{count}[j, h_j(i)] \right) \]

This estimator has the following proven guarantee [6]: The estimated value is at least as big as the real value ($a_i \leq \hat{a}_i$), and with a probability of at least $1 - \delta$, the bound $\hat{a}_i \leq a_i + \epsilon \sum_{k=0}^{n} (a_k)$ holds. For the sketch parameters $\epsilon$ and $\delta$, the following relation holds: $w = \lceil \frac{\epsilon}{\delta} \rceil$ and $d = \lceil \ln \frac{1}{\delta} \rceil$.

There are two options to estimate the frequency of a range of items. The naïve way is to simply sum up the point estimates in this range. The problem is that the error guarantee and the time cost increases linearly with the length of the range. Another solution is to use a hierarchical CM [6, 5]. But this increases the complexity and the update time. The current implementation uses the naïve way. But, in order to bound the time costs, it has a budget of $2 \cdot k$. Point estimates are done for the lowest and the highest $k$ values in the interval. The values in between are estimated as the average from the current seen values. This method has no formal error bound. This solution was chosen because we do not want to increase the update costs, nor have unbounded range estimation costs.

3.3.2 The FastAGMS Sketch

The FastAGMS Sketch [4] is nearly the same as the CountMin sketch. The data structure and the hash functions are the same. Additionally, it has $d$ functions $\xi : \{0...n\} \rightarrow \{-1, +1\}$ taken uniformly at random from a 4-wise independent $\pm 1$
random variable family. For the generation of the $\pm 1$ random variables the extended Hamming scheme described in [23] is used. The count operation is now defined as:

$$\forall j \in \{0...d-1\} : \text{count}[j, h_j(i)] \leftarrow \text{count}[j, h_j(i)] + \xi_j(i)$$

To estimate the frequency of an item the median is taken instead of the minimum:

$$\hat{a}_i = \text{median}_{j \in \{0...d-1\}} (\text{count}[j, h_j(i)] \cdot \xi_j(i))$$

The median estimator has the following proven guarantee [4]: With a probability at least $1 - \delta$ it holds $\hat{a}_i \in \left( a_i \pm \epsilon \sqrt{\sum_{k=0}^{n}(a_k)^2} \right)$. For the sketch parameters $\epsilon$ and $\delta$, it holds that $w = \lceil \frac{1}{\epsilon^2} \rceil$ and $d = \lceil \ln \frac{1}{\delta} \rceil$.

To estimate the frequency of a range of items, we can do the same considerations as for the CountMin sketch.

3.3.3 Optimizations

- The used pairwise independent hash family functions are defined as $h(x) = ax + b \mod p$ where $p$ is a prime (see also [17]). $a$ and $b$ are for each hash function chosen uniformly at random at random at the sketch construction time. Because modulo by a prime is an expensive operation, the mersenne trick is applied: if $p$ is chosen as a mersenne prime, i.e. there exists a natural number $s$ such that $p = 2^s - 1$ holds, then $k \mod p$ can be computed as: $(k \& p) + (k \gg s)$. Where $\&$ denotes the bitwise-AND and $\gg$ denotes the logical right shift.

- If the width $w$ of the counter array is padded such that $\exists s \in \mathbb{N} : w = 2^s$ holds, one can also replace a modulo operation by a shift: $\text{counter}[j, h(i) \mod w] = \text{counter}[j, h(i) \gg s]$. Note that the hash function is needing a modulo with a prime and that the hash returned by the function is in general not in $\{0..w-1\}$, which makes the additional modulo $w$ necessary.

3.4 Evaluation

This section does not evaluate how Crescando performs with the improved selectivity estimates, which will be covered in the overall evaluation chapter 6. Instead, it evaluates the two sketches in terms of accuracy and speed.

3.4.1 Setup

Two micro benchmarks are used to evaluate the sketches and executed on different workloads.
1. **Accuracy benchmark:** This benchmark counts with a sketch all elements in a workload. After the counting, all element frequencies are estimated and compared with the exact values from the workload. Each test is run ten thousand times. The width is set to the number of distinct values in the workload divided by the depth. Additionally, the width padding described in 3.3.3 is not used. This guarantees for all tests fair and comparable space usages.

2. **Speed benchmark:** This benchmark measures the time used to call the *count* respectively the *pointEstimate* operation ten million times. This is done twenty times. The values are chosen uniformly at random from a real workload\(^{(2)}\). The width is computed the same way as in the accuracy test.

3. **Zipf workload:** The accuracy benchmark uses a zipf distribution. The zipf parameter is varied from zero up to three and the values are in the range \[0, 999\] resulting in \(N = 1000\) distinct values. As a reminder, the probability of a value with the rank \(k\) is computed as \(\Pr(k; s, N) = \frac{1/k^s}{\sum_{n=1}^{N} (1/n^s)}\). Because the test workload should not contain any values with a zero count, this probability is multiplied with the inverse probability of the item with the highest rank. Formally: \(\text{Count}(k; s, N) = \Pr(k; s, N) \cdot \frac{1}{\Pr(k; s, N-1)}\)

4. **Error computation:** The error is computed as \(\sum_{i=0}^{n} |a_i - \hat{a}_i|\) which is the \(L_1\) norm and then divided by \(n\) (which is 1000 for the zipf workload) to get the average. Finally, it is divided by \(\sum_{i=0}^{n} a_i\) to get the relative error.

### 3.4.2 Depth Parameter

The depth parameter has an influence on accuracy and, unlike the width, also on the update cost. The smaller the depth is, the less counters have to be updated. But choosing it too small results in a bad confidence for the bounds. For the same space usage, a good depth gives a high accuracy on different workloads. To find the best depth, the accuracy benchmark is run with different depths on the zipf workload.

As we can see in figure 3.2, choosing \(d = 3\) seems a good choice for CountMin. Only for distributions with a high skew the error still decreases for a higher depth. Choosing \(d > 5\) results in a poorer accuracy for all tested workloads\(^{(3)}\).

Figure 3.3 shows the same benchmark results for the FastAGMS. We notice that the accuracy is worse for even depths. This is due to the median estimator. In the case of \(d = 2\) it actually degenerates to a mean estimator. Also, for distributions with a low skew, the depth actually has no influence on the accuracy. For high skews, choosing a higher depth results in more accurate estimates. Ignoring again the two highest skewed distributions, choosing \(d = 3\) seems to be also a good update-cost accuracy tradeoff.

\(^{(2)}\)The tests uses attribute distributions provided by Amadeus. See section 6.1.2.
\(^{(3)}\)recall that the width is inversely proportional to the depth
Figure 3.2: Accuracy of the CountMin sketch for different depths

Figure 3.3: Accuracy of the FastAGMS sketch for different depths
3.4 Evaluation

3.4.3 Accuracy

The accuracy depends also on the distribution of the data. To make a comparison between the two sketches, the accuracy tests are run again with the zipf workload. For both sketches the depth is set once to three and once to seven. The results are comparable with the ones from [24].

![Graph (a) d = 3](image1)
![Graph (b) d = 7](image2)

Figure 3.4: Accuracy for different depths

As figure 3.4 shows, for data with a high skew CountMin is more accurate, and for a low skew the FastAGMS is more accurate. Note that with a higher depth, the intersection point of the two curves slightly shifts to the right.
3.4.4 Speed

The speed benchmark was run on integer items as well as on string items. The values two, three, nine and fifteen were chosen as the string length. We are comparing the speed of CountMin and FastAGMS with and without padding. Additionally the speed of an naïve implementation was measured. This implementation uses a hash map provided by the GNU C++ compiler.

Figure 3.5: Speed of the different sketches with $d = 3$

Figure 3.5 shows the different times for the *count* respectively the *pointEstimate* operations. As expected, FastAGMS is slower for the count operation and considerably slower for the pointEstimate operation. We can also see that padding gives a considerable speed up. The padded CountMin is about two times faster than the naïve implementation for counting string items and nearly four times faster at counting integer items. FastAGMS is more than two times faster at counting integer types than the naïve. Due to a bad implementation of the $\xi$ function for strings, FastAGMS is at counting strings a bit slower than the naïve. Because of the median estimator, FastAGMS is at estimating of any typ slower than the naïve\(^{(4)}\).

Typically, a *DetailedStatisticsSelect* needs to count values from about one million records. At update speed of 100ns, executing it needs then 100ms, which is acceptable. But because the naïve implementation uses much more space, the sketches are preferable.

\(^{(4)}\) Again, performance depends on bad implemented $\xi$ functions for strings
Chapter 4

Cost Model

This chapter shows how the cost estimation of a Index Union Join Plan 2.3 can be improved by extending the cost model to factor in lookup costs for different predicate index types.

With a precise cost estimation of a Index Union Join Plan (see section 2.3) the optimizer can choose the best plan. Although, it does not seem realistic to get a precise model, the aim should be to get cost estimates as accurate as possible. Improving the selectivity estimation does already increase the precision. However, the current cost model, which is embedded in the baseline optimizer (see section 2.6), only considers the selectivity of indexed predicates. For example, adding a new empty index does not influence the cost of the plan. Obviously, the costs of the plan should be higher because an index has lookup costs, even if it is empty. The baseline optimizer uses a magic threshold that prevents it from building too many indexes. Also the lookup costs are different depending on the index type. For example, if there are no range queries, the baseline optimizer always prefers a MultiHashMap to an OneDimRTree (see section 2.4), although in some cases a OneDimRTree would be cheaper.

We will first transform the embedded cost model of the baseline optimizer to an explicit one with a clear interface. We develop then a new cost model that respects the lookup cost and uses the same interface.

4.1 Design

The abstract CostModel has two operations: getCost(in IS : IndexSet) : Cost and getIndexCost(in i : Index) : Cost. The getCost function sums up the index costs from the IndexSet IS. An Index i is a set of queries Q_i, which has some properties: a_i denotes the attribute of the index i, Type_i denotes the index type such as MultiHashMap or JaggedArray, |Q_i| is the number of queries and avgSel(a_i, Q_i) the average selectivity. An actual CostModel can compute the costs with these properties. We want to emphasize here that in the implementation the Index construct as used in this thesis does not exists. Instead, the implemen-
tation uses in the optimization context a lightweight object to represent an index. The actual index data structures are built from these lightweight objects. The unindexed queries $Q_{UI}$ are handled transparently: They are seen as another index with the index type $Type_{UI}$. Their average selectivity is always one because they are executed on each record. This model is now expressed in more formal way:

$$\text{Cost(IS)} := \sum_{i \in IS} \text{IndexCost}(Type_i)(i)$$  \hspace{1cm} (4.1)

To express that the $\text{IndexCost}$ function may be completely different depending on the type of the index, we write $Type_i$ between $\langle \rangle$. Parameters between such angles can be seen as compile-time parameters in contrast the run-time parameters given in parentheses $()$.

### 4.2 The Baseline Cost Model

The baseline optimizer has its cost model embedded in its algorithm. This section shows how it can be expressed explicitly using the described design. This way it can be easily compared with other cost models. Additionally, we have to modify the baseline optimizer to a version that uses the $BaselineCostModel$. This optimizer is the so called $GreedyOptimizer$ shown in algorithm 2.

The baseline optimizer considers the best index as the one with the highest gain. In the case of costs, instead of a gain, the best index would be the one with the lowest cost per operation in it. Given the $rangeGain$ function from section 2.6 this can be formulated as:

$$\min_{\text{BaselineIndexCost}(Type_i)(i)/|Q_i| \{i \in Indexes\} := \max_{\text{RangeGain}[a]} \{a \in A\}}$$

The baseline optimizer has also a threshold $\text{thresh}$, which is used to determine when no more indexes should be built. Such a threshold should belong to the cost model. Because $Q_{UI}$ is treated also as an Index, the new optimizer can stop, when each query was assigned to one Index. As there is no $rangeGain$ defined for the unindexed queries, these cost have to be computed differently. The cost model has to distinguishes between unindexed $Type_{UI}$ and indexed $Type_{Indexed}$; it does not distinguish between different attribute types nor index types. The aim is now to express the cost function in terms of the old gain function and the values provided by an Index. If the $rangeGain$ is multiplied by $-1$, the maximum gain has now the smallest value. It is important to notice that because the cost based optimizers utilize relative costs to find the currently best index, the gain has to be multiplied
4.2 The Baseline Cost Model

by $|Q_i|$.

\[
\text{BaselineIndexCost}_{\text{Type Indexed}}(i) / |Q_i| = -1 \cdot \text{RangeGain}(a)
\]

\[
\text{BaselineIndexCost}_{\text{Type Indexed}}(i) = -1 \cdot \text{RangeGain}(a) \cdot |Q_i|
\]

\[
= -1 \cdot \sum_{a \in Q_i} (1 - \text{selectivity}(a, q)) \cdot |Q_i|
\]

\[
= -1 \cdot |Q_i| \cdot (1 - \text{avgSel}(a_i, Q_i)) \cdot |Q_i|
\]

Note that the cost function has a quadratic factor $|Q_i|$. Now the same transformation \((-1 \cdot |Q_i|)\) is applied to the threshold giving us the cost for unindexed queries:

\[
\text{BaselineIndexedCost}_{\text{Type UI}}(i) = -1 \cdot \text{thresh} \cdot |Q_{UI}|
\]

4.2.1 GreedyOptimizer

Algorithm 2: GreedyOptimizer

| Input: QuerySet QS, CostModel CM, IndexSet PI; |
| Output: IndexSet AI ← ∅, QuerySet Q_{UI} ← ∅; |

repeat

\( \text{foreach Index idx ∈ PI do} \) // Refresh possible indexes

\( \text{foreach Query q ∈ QS do} \)

\( \text{if q isIndexableBy idx then} \) \( \text{idx ← idx ∪ q;} \)

end

\( \text{Index } i ← \min_{CM, \text{IndexCost}(i)} / |Q_i| \{ i \in PI \}; \) // Take best index

if \( CM = \text{BaselineCostModel} \) then

\( \text{if Type}_i = \text{OneDimRTree ∧ (∃) \in PI : a_i = a_j ∧ Type}_j = \text{MultiHashMap} \) then

\( i ← \); 

end

\( \text{if Type}_i = \text{Type}_{UI} \) then

\( Q_{UI} ← QS; \)

\( QS ← ∅; \)

else

\( AI ← AI ∪ i; \)

\( PI ← PI \setminus i; \)

\( QS ← QS \setminus Q_i; \)

end

until \( QS = ∅; \)

Algorithm 2 illustrates the new GreedyOptimizer. As input it gets the query set QS with the queries to index, a cost model CM and a set of all possible indexes PI (This set contains also an ‘index’ for the unindexed queries). The algorithm returns then an IndexSet AI that denote the active indexes\(^{(1)}\) and a query set Q_{UI}, which denote the unindexed queries in the final plan. Like the baseline optimizer, the GreedyOptimizer decides at each iteration to build the ‘best’ index, but here the best index is determined by the given CostModel CM.

\( ^{(1)}\)These indexes are then later really built to form an actual index union join plan.
The baseline optimizer prefers a MultiHashMap to a OneDimRTree. As this cannot be modeled in the new BaselineCostModel, the GreedyOptimizer has special code, if CM happens to be the BaselineCostModel.

### 4.3 The Index-aware Cost Model

This cost model is based on the computational cost of executing a plan for one record. It distinguishes between the unindexed queries and all three different indexes. The index costs consist of two parts, the evaluation cost $\text{cost}_{\text{eval}}$ and the lookup cost $\text{cost}_{\text{lookup}}$.

$$\text{IdxAwareIndexCost}(\text{Type}_i)(i) = \text{cost}_{\text{lookup}}(\text{Type}_i)(i) + \text{cost}_{\text{eval}}(i)$$

As the cost function $\text{cost}_{\text{eval}}$ does not depend on the index type, it has no parameter $\text{Type}_i$. Evaluating a query, which means testing whether all predicates match a record, is considered to be a constant cost $\text{c}_{\text{eval}}(\text{Type}_i, \text{Type}_a)$. This constant depends on the index type $\text{Type}_i$ as well as on the type of the attribute $a_i$, denoted as $\text{Type}_a$. A query in an index is evaluated with a probability equal to its selectivity:

$$\text{cost}_{\text{eval}}(i) = \sum_{q \in Q_i} \text{sel}(q) \cdot \text{c}_{\text{eval}}(\text{Type}_i, \text{Type}_a)$$

$$= \text{avgSel}(a_i, Q_i) \cdot |Q_i| \cdot \text{c}_{\text{eval}}(\text{Type}_i, \text{Type}_a)$$

Because the unindexed 'Index' has always an average selectivity of one and there are no lookup costs for unindexed queries, the costs can be computed as:

$$\text{IdxAwareIndexCost}(\text{Unindexed})(i) = |Q_{UI}| \cdot \text{c}_{\text{eval}}(\text{Type}_{UI})$$

The lookup costs of the various index types are defined in section 4.5. The next subsection introduces the setup to determine constants that are used in this cost model.

#### 4.3.1 Constant Initialization

There are many constants in the index-aware cost model. There is the evaluation constant for each index and attribute type $\text{c}_{\text{eval}}(\text{Type}_i, \text{Type}_a)$ \(^\text{(2)}\) and there are constants for index lookups. The constants are computed during the initialization of Crescando. At that point, the schema is loaded, but there is currently no data and the scan threads are not started yet. The index constants should be computable in a way such that adding an index would require minimal changes in the existing code. There are three execution-time measurements done for each possible $(\text{Type}_i, \text{Type}_a)$ pair. Each measures the time that is needed to execute a synthetic workload $Q_{\text{syn}}$ on synthetic data with an index. The three measurements are $M_{\text{never}}$, $M_{\text{always}}$, and $M_{\text{some}}$. The data and the workload are chosen for each

\(^{\text{(2)}}\)Some combinations are illegal such as a $\text{c}_{\text{eval}}$(JaggedArray, String).
measurement in a way such that \( M_{\text{never}} \) never evaluates a query, \( M_{\text{always}} \) evaluates always each query and \( M_{\text{some}} \) has to evaluate \( \text{sel}_{\text{some}} \cdot |Q_{\text{syn}}| \) queries. With these data points, a system of linear equations can be solved, which yields the cost constants pertaining to the index type in question. To compute \( c_{\text{eval}}(\text{Unindexed}) \) a special measurement is used. Summarizing, the index constants have to be computable with the three measured execution times, \( \text{sel}_{\text{some}} \) and \( |Q_{\text{syn}}| \).

## 4.4 Related Work

The computational cost for searching a key in a hash map have been studied for a long time. One of the first analysis for hashing with linear probing originates from Donald Knuth [15] in 1963. He gave later in [16] a detailed analysis of hashing with linear probing. In recent years linear probing has become more popular since it has a good cache behavior. Knuth assumes for his analysis a truly random hash function, [22] presents hash functions for linear probing and presents a more accurate analysis for those functions. In this thesis, we assume that the actual hash function is unknown and use therefore Knuth’s analysis.

The OneDimRTree and its construction originates from [7]. The paper [27] gives an analysis for R-Trees in general. The presented cost analysis is meant for optimization purposes, and requires therefore only a small set of meta data to give an estimate.

## 4.5 Index Costs

In this section we show how the lookup costs for each index is computed and how the constants for this index are determined.

### 4.5.1 JaggedArray

#### Lookup Cost

The lookup cost for the JaggedArray JA is in \( O(1) \) and can be expressed with a constant \( c_{\text{lookup}}(\text{JA}, \text{Type}_a) \). Using the formula for the evaluation cost (4.2), the index cost can be computed as:

\[
\text{IdxAwareIndexCost}(\text{JA}, \text{Type}_a)(i) = c_{\text{lookup}}(\text{JA}, \text{Type}_a) + \text{avgSel}(a_i, Q_i) \cdot |Q_i| \cdot c_{\text{eval}}(\text{JA}, \text{Type}_a)
\]

#### Constants

Because there are no evaluation costs in the case of \( M_{\text{never}} \) one has:

\[
c_{\text{lookup}}(\text{JA}, \text{Type}_a) = M_{\text{never}}
\]
The evaluation cost constant is then computed as:
\[
    c_{\text{eval}}(JA, \text{Type}^a) = \frac{M_{\text{always}} - c_{\text{lookup}}(JA, \text{Type}^a)}{|Q_{\text{syn}}|} + \frac{M_{\text{some}} - c_{\text{lookup}}(JA, \text{Type}^a)}{\text{sel}_{\text{some}}|Q_{\text{syn}}|}
\]

### 4.5.2 MultiHashMap

#### Lookup Cost

Recall that hash collisions are resolved with linear probing and that queries having the same predicate are collected in a list (see section 2.4). Let us denote with \(|\mathcal{X}|\) the number of different predicates, in other words the number of different keys. The lookup costs of a MultiHashMap \(HM\) is computed in terms of the number of probes needed to find a key. In the case of an unsuccessful search this number is higher than in the case of a successful search. Formally:

\[
    \text{cost}_{\text{lookup}}(HM, \text{Type}^a)(i) = \Pr[\text{success}] \cdot \text{avgProbesSuccess} \cdot c_{\text{successLookup}}(HM, \text{Type}^a) + (1 - \Pr[\text{success}]) \cdot \text{avgProbesNosuccess} \cdot c_{\text{nosuccessLookup}}(HM, \text{Type}^a)
\]

Let \(|HM|\) denote the size\(^{(3)}\) of the hash map. The load factor \(\alpha\) of \(HM\) is defined as \(|\mathcal{X}|/|HM|\). The number of probes can be estimated according to D. E. Knuth’s [16] analysis:

\[
    \text{avgProbesSuccess} \approx \frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2}\right)
\]

\[
    \text{avgProbesNosuccess} \approx \frac{1}{2} \left(1 + \frac{1}{1 - \alpha}\right)
\]

\(|\mathcal{X}|\) can be expressed as \(|Q_i| - \text{noDuplicates}\). To give an estimate for the number of duplicated keys, we make an assumption:

**Assumption**: The values in query predicates have a similar distribution as the values in the table.

The number of duplicated keys is now estimated as \(\text{avgSel}(a_i, Q_i) \cdot |Q_i|\). Thus,

\[
    |\mathcal{X}| = |Q_i| - \text{avgSel}(a_i, Q_i) \cdot |Q_i| \quad (4.4)
\]

The probability of a successful lookup remains to be derived. If we assume that there are no two queries with the same key in the index, then a good estimation would be the same as for evaluating a query: \(\text{avgSel}(a_i, Q_i) \cdot |Q_i|\). Because there are duplicated keys, we have to subtract them: \(\text{avgSel}(a_i, Q_i) \cdot (|Q_i| - \text{noDuplicates})\). We can reuse equation (4.4), but because it is a crude estimate the whole expression can be higher than 1. Therefore, the probability is estimated as:

\[
    \Pr[\text{success}] \approx \min(\text{avgSel}(a_i, Q_i) \cdot (|Q_i| - \text{avgSel}(a_i, Q_i) \cdot |Q_i|), 1) \quad (4.5)
\]

\(^{(3)}\)With the size we mean the maximum number of different keys that \(HM\) can store
4.5 Index Costs

Constants

For the measurement $M_{\text{never}}$ the equation (4.3) can be reduced to:

$$M_{\text{never}} = c_{\text{nosuccessLookup}}(HM, Type_a)$$

This is because $Pr[\text{success}]$ is obviously zero, there are no evaluation costs, and $\text{avgProbesNosuccess}$ is one\(^{(4)}\). For the two other measurements this analysis makes two observations. The first is that there are no hash collisions in the case of the synthetic workload, which has values in $\{1, \ldots, n\}$. This is because for primitive types the hash function is defined as $h(x) = x \mod |HM|$, $x$ is in $\{1, \ldots, n\}$ and $|HM| > 8 \cdot n$.\(^{(5)}\) The second is that $Pr[\text{success}] = 1$. For $M_{\text{always}}$ this is trivially true. For $M_{\text{some}}$ this is true because of (4.5) and $|HM| > 8 \cdot n$. With these properties and (4.3) follows the equation system:

$$M_{\text{always}} = c_{\text{successLookup}}(HM, Type_a) + |Q_{\text{syn}}| \cdot c_{\text{eval}}(HM, Type_a)$$

$$M_{\text{some}} = c_{\text{successLookup}}(HM, Type_a) + |Q_{\text{syn}}| \cdot sel_{\text{some}} \cdot c_{\text{eval}}(HM, Type_a)$$

4.5.3 OneDimRTree

Lookup Cost

The lookup cost of the OneDimRTree $RT$ consists of the cost for visiting inner nodes $IN_v$ and visiting leaf nodes $LN_v$. Let $|IN_v|$ denote the expected number of visited internal nodes and $|LN_v|$ denote the one of the leaf nodes.

$$\text{cost}_\text{lookup}(RT, Type_a)(i) = |IN_v| \cdot c_{\text{InNodeLookup}}(RT, Type_a)$$

$$+ |LN_v| \cdot c_{\text{LfNodeLookup}}(RT, Type_a)$$

(4.6)

The cost of visiting a node is assumed to be constant. The difficult part is to estimate the amount of visited nodes. The analysis is based on [27]. Let $h$ be the height of the tree and let the root be at level $h$ and the leaf nodes at level 1 (the queries are then at “level 0”). The total expected number of nodes accessed $|NA_v|$ is given as $\text{(6)}$:

$$|NA_v| = \sum_{l=1}^{h} (D_l)$$

$D_l$ is the density $D$ at a tree level $l$. $D$ has the following definition in [27]:

**Definition 4.5.1.** The density $D$ of a set of $N$ rectangles with average extent $s = (s_1, \ldots, s_d)$ is the average number of rectangles that contain a given point in $d$-dimensional space.

\(^{(4)}\)All queries have the same value and all records have the same value, which do not produce a hash collision

\(^{(5)}\)For other types like strings this might not be true, but as the evaluation shows there are no significant differences between integers and strings

\(^{(6)}\)The parameter $q$ given in [27] disappears, because Crescando has only point queries to answer
In our case, where the dimension $d$ is 1, the density at level 0 is simply $\text{avgSel}(a_i, Q_i)$. \[ |Q_i| \] gives the following recursive formula to compute density at a level:

$$D_{l+1} = \left(1 + \frac{D_l^{1/d} - 1}{f^{1/d}}\right)^d$$

Where $f$ denotes the fan-out of the tree. Because $d$ is 1 in our tree we got finally:

$$|LN_v| = D_1 = 1 + \frac{D_0 - 1}{f} = 1 + \frac{\text{avgSel}(a_i, Q_i) \cdot |Q_i| - 1}{f}$$

And:

$$|IN_v| = \sum_{l=2}^{h} (D_l)$$

In the OneDimRTree the fan-out $f$ is a compile-time parameter of the tree and the height can be easily computed in terms of $|Q_i|$ and $f$.

### Constants

The synthetic workload for the measurements contains no range queries. Contrary to expectations, this is actually an advantage for computing the constants. To compute them, one has again to estimate the number of visited nodes, which is much easier if there are no range queries. In the case of the never measurement, a lookup has only to visit the root node:

$$c_{\text{lnNodeLookup}}(\text{RT}, \text{Type}_a) = M_{\text{never}}$$

In the case of $M_{\text{always}}$, the lookup visits all nodes, because there is a leaf entry for each query and because all records matches all queries. Let $|N_l|$ denote the number of nodes at level $l$, then:

$$|N_0| = |Q_{\text{syn}}| \quad (4.7)$$

$$|N_l| = |N_0|/f^l$$

In the case of $M_{\text{some}}$, the number of nodes $|NV_l|$ to be visited at a level $l$ can be given as:

$$|NV_0| = \text{sel}_{\text{some}} \cdot |Q_{\text{syn}}| \quad (4.8)$$

$$|NV_l| = |NV_0|/f^l$$
The number of leaf nodes accessed is then $|\text{NV}_1|$. By applying (4.6), (4.7) and (4.8) the following equation system can be derived:

\[
M_{\text{always}} = \sum_{l=2}^{h} (N_l) \cdot c_{\text{lnNodeLookup}}(RT, \text{Type}_a)
+ |N_l| \cdot c_{\text{lfNodeLookup}}(RT, \text{Type}_a)
+ |Q_{\text{syn}}| \cdot c_{\text{eval}}(RT, \text{Type}_a)
\]

\[
M_{\text{some}} = \sum_{l=2}^{h} (N\text{V}_l) \cdot c_{\text{lnNodeLookup}}(RT, \text{Type}_a)
+ |\text{NV}_l| \cdot c_{\text{lfNodeLookup}}(RT, \text{Type}_a)
+ |Q_{\text{syn}}| \cdot \text{sel}_{\text{some}} \cdot c_{\text{eval}}(RT, \text{Type}_a)
\]

### 4.6 Design Notes

In the old system, adding a new index type would require to modify the baseline optimizer and using some heuristic so that this optimizer can use the index. With the new cost model, adding an index to Crescando requires only minor changes. The new index has to implement two functions: one to initialize constants with the mentioned measurements and one to estimate costs. In order that an optimizer considers a new index, the index has to be added to the input set containing the possible indexes. New plan enumeration algorithms should not contain any heuristics such as the one in the GreedyOptimizer for the BaselineCostModel.

Adding a new cost model is also simple. A new cost model has to inherit from a provided abstract cost model and has to implement at least the `getIndexCost` function. The new cost model can be used immediately with any of the novel plan enumeration algorithms.

### 4.7 Evaluation

This section evaluates the new cost model in terms of accuracy and compares it with the baseline cost model. Later, in chapter 6 the performance of Crescando with the new cost model is evaluated.

#### 4.7.1 Setup

To measure the accuracy we run two benchmarks on Crescando. The improved selectivity estimation described in the previous chapter was active for all measurements.

- **Schema & Workload:** Both benchmarks were run with a synthetic workload on the Amadeus schema and its data distribution. A detailed description of this synthetic workload and on the Amadeus schema can be found in the
section 6.1, but for now the information provided here should be enough. The Amadeus schema is chosen because it provides many different attribute types and the data has a distribution taken from a real application. Other than in the constants measurement setup, both the data and the workload is not uniformly distributed.

- **Accuracy measurement:** The benchmarks measure two values for each scan: the time taken to execute a plan and its computed cost. Each value pair is then plotted as a point in a plane. For a perfect cost estimation, the actual time taken to execute a plan is equal to the estimated cost multiplied by a factor. This means, that these points should lie on a straight line passing through the origin.

- **Single index benchmark:** To evaluate only one index, a plan is created with only that index. Furthermore, queries that do not fit into this index are filtered out. The workload is slowly increased in order to get different values. At some point, the benchmark creates more queries than Crescando can handle. Measurement values in such an overloaded system are discarded.

- **Mixed benchmark:** To measure the accuracy in the presence of different indexes, Crescando is run with the new GreedyOptimizer. This benchmark is then executed once with the baseline cost model and once with the index-aware cost model.

### 4.7.2 Accuracy of JaggedArray

The accuracy of the JaggedArray was measured on a boolean attribute. In figure 4.1 we can see the described measurement points. Furthermore, all the figures in this section have a trend line. The $R^2$ of the line tells us how accurate the model is. For all the different indexes, the factor given in the equation of the line should have about the same value. Here, the equation is $y = 0.2274 \cdot x$ and the mentioned factor is 0.2274. A similar factor is required in order to give good cost estimates for several indexes at the same time. As one can see in the figure, the all the measurement points lie closely on the trend line, which promises a good accuracy.

![Figure 4.1: JaggedArray accuracy](image)
4.7 Evaluation

4.7.3 Accuracy of MultiHashMap

The accuracy for the MultiHashMap was measured once on an integer attribute and once on a string attribute. Figure 4.2 shows the results for both attributes. In the two figures, the measurement points lie again closely to the trend lines. In the case of the string attribute, one can notice some non linear factor for execution times around one second. Another problematic result is that the factors of the trend lines are noticeable higher than the one of the other index types.

\[
y = 0.2873x \\
R^2 = 0.99458
\]

\[
y = 0.2722x \\
R^2 = 0.98624
\]

Figure 4.2: Cost-model accuracy for the MultiHashMap
4.7.4 Accuracy of OneDimRTree

Figure 4.3 shows the accuracy of the cost estimation for the OneDimRTree. The benchmark was run on an integer attribute with a workload having range queries. In comparison to the other cost estimations, there are many outliers in this figure. Points that are below the trend line are points where the cost model underestimates the cost of an index. Underestimating is worse than overestimating as explained earlier in the chapter 3. It seems that underestimating long-range queries causes this error.

For example, there is a query with a long range and many other point queries in this range. An OneDimRTree with these queries has then at each node level an interior node with this long range. Additionally to the normal search path, searching queries with a value in this range will require to always following these interior nodes.

Figure 4.3: OneDimRTree accuracy
4.7 Evaluation

4.7.5 Accuracy for Multiple Indexes at the Same Time

In the figure 4.4 we can see that the index-aware cost model has definitely the higher accuracy than the baseline cost model. Plans that take more time have in general more queries in it. More queries mean more indexes, which in turn means that cost estimation errors are summed up. It is believed that heavily underestimated plan costs originate from cache trashing that is ignored in the new cost model. The cost estimates for the baseline cost model are negative and seem to have a non-linear factor. I assume, that this is mainly caused by the $|Q_i|^2$ part of the computation formula.

![Figure 4.4: Accuracy for the index-aware (left) and baseline (right) cost model](image-url)

$y = 0.1557x$

$R^2 = 0.88539$

$y = -118595x$

$R^2 = 0.56789$
Chapter 5

Plan Enumeration

With a cost model described in the last chapter, an optimizer is able to compare plans. If it can pick the cheapest plan out of all possible plans, the execution time would be as small as possible. However, finding the optimal plan is a hard problem and in our case the search space is by far too big to do an extensive search (see section 5.1). Also, time spent for optimization is time lost for execution. The current optimizer greedily builds a plan (section 4.2.1) in a relatively short time, but these plans are often suboptimal. To find a better optimizer, we will first analyze the optimization problem and then suggest new algorithms.

5.1 The Optimization Problem

This section classifies the Index Union Join optimization problem (IUJO). Recall that the Index Union Join Plan (see section 2.3) consist of a set of indexes and a set of unindexed queries. First, we describe Exact Cover (EC) that is an NP-complete problem [14]. Then, we transform this decision problem into an optimization problem, which we call Minimum Exact Cover (MIN-EC). Finally, we show that IUJO is at least as hard as MIN-EC.

5.1.1 Exact Cover

Given a Collection $C$ of subsets of a set $W$, an exact cover of $W$ is a sub-collection $C'$ of $C$, satisfying:

- The intersection of any two subsets of $C'$ is empty: $\{x \in W \mid x \in S_i \land x \in S_j \land S_i \in C' \land S_j \in C'\} = \emptyset$

- The union of all subsets of $C'$ is the whole set $W$: $\cup_{S_i \in C'} = W$

The decision problem Exact Cover (EC) asks if such an exact cover $C'$ for a given $C$ and $W$ exists. As already mentioned, Exact Cover is known to be NP-complete [14].
Example Let $W = \{1, 2, 3\}$ and $C = \{\{2\}, \{1, 3\}, \{1, 2, 3\}\}$ then an exact cover would be $C'_1 = \{\{1, 2, 3\}\}$ or $C'_2 = \{\{2\}, \{1, 3\}\}$ but no other subsets of $C$.

5.1.2 Minimum Exact Cover

Minimum Exact Cover (MIN-EC) is the optimization problem of finding an exact cover $C'$ such that its cost is minimal.

Definition 5.1.1. \(MIN-EC = (L, M, \text{cost}, \text{goal})\) be the minimum exact cover optimization problem, with\(^1\):

- \(L\) denotes the language of all valid inputs. Here it is \((W, C, w_C) \in V \times \{C \subseteq \mathcal{P}(W)\} \times \mathbb{R}^{|C|}\) \(^2\) \(w_C\) denotes a set of weights, where each weight belongs to one subset $S \in C$.

- \(M\) is a function of \(L\) such that for each $x \in L$, $M(x)$ is the set of permitted solutions for $x$. This is the set of all $C'$, where $C'$ is an exact cover of $(W, C)$.

- \(\text{cost}\) is the sum of the weights of all subsets $S \in C'$: $\sum_{S \in C'} w_{CS}$.

- \(\text{goal}\) is in this case the Minimum function.

A permitted solution $\alpha \in M(x)$ is called optimal for $x \in L$ of the optimization problem $MIN-EC$, iff

$$\text{cost}(\alpha, x) = \text{Opt}_{MIN-EC}(x) = \min \{\text{cost}(\beta, x) \mid \beta \in M(x)\}$$

Example Let $W = \{1, 2, 3\}$, $C = \{\{2\}, \{1, 3\}, \{1, 2, 3\}\}$ and $w_C = \{1, 1, 4\}$ then an exact cover would be $C'_1 = \{\{1, 2, 3\}\}$ or $C'_2 = \{\{2\}, \{1, 3\}\}$ but as $\text{cost}(C'_1) = 4 \geq 2 = \text{cost}(C'_2)$ the optimum would be $C'_2$.

5.1.3 NP-Hardness of Minimum Exact Cover

First, we show that Minimum Exact Cover is an \textit{NP-optimization problem} (NPO) and then we proof that it is \textit{NP-hard}. According to the definition of NPO in [12, Def. 6.12.], three conditions have to be met:

1. It can be determined in polynomial time, if an input $x$ is in the language $L$.

2. The size of every solution $y \in M(x)$ is polynomial bounded in the size of $x$ as well as it can be determined in polynomial time if a $y$ belongs to $M(x)$.

3. The function \(\text{cost}\) is polynomial-time computable.

Lemma 5.1.2. Minimum Exact Cover is in NPO.

\(^1\)We reusing a slightly simplified version of the definition of an optimization problem provided in [12, Def. 2.13.]

\(^2\)\(V\) is the von Neumann universe and $\mathcal{P}$ the power set operation
5.1 The Optimization Problem

Proof. As one can easily see, condition (1) is met (verify that \( C \) describes a collection of subsets of \( W \) is in \( O(|W| \cdot |C|) \)). (2) is also true because for an \( C' \in M(x) \) holds that \( C' \subseteq C \). Finally, (3) is obviously also true (linear in \( |x_C| \)). □

The definition of NP-hard is given in [12, Def. 6.14.] as:

Definition 5.1.3. Let \( U = (L, M, \text{cost, goal}) \) be an optimization problem in NPO. The Threshold-Language of \( U \) (\( \text{Lang}_U \)) is:

\[
\text{Lang}_U = \{ (x, a) \in L \times \mathbb{R} \mid \text{Opt}_U(x) \leq a \}
\]
if \( \text{goal} = \text{Minimum} \), and

\[
\text{Lang}_U = \{ (x, a) \in L \times \mathbb{R} \mid \text{Opt}_U(x) \geq a \}
\]
if \( \text{goal} = \text{Maximum} \).

\( U \) is said to be NP-hard if \( \text{Lang}_U \) is NP-hard.

Theorem 5.1.4. Minimum Exact Cover is NP-hard.

Proof. According to lemma 5.1.2 MIN-EC is in NPO, which allows us to apply definition 5.1.3: it is sufficient to show that \( \text{Lang}_{\text{MIN-EC}} \) is NP-hard. We show that Exact Cover is polynomial-time reducible [12, Def. 6.10.] to \( \text{Lang}_{\text{MIN-EC}} \): \( \text{EC} \leq_p \text{Lang}_{\text{MIN-EC}} \).

Let \( x = (W, C) \) be an element of the language \( \text{L}_{\text{EC}} \); that is, there exists an exact cover \( C' \) for \( W \) and \( C \). Then let \( w_C \) be equal to one for each subset of \( C \) and let \( m = |C| \) be the number of subsets in \( C \). Then it is clear that:

\[
(x, w_C, m) \in \text{L}_{\text{MIN-EC}} \iff x \text{ has an exact cover.}
\]

Because generating \( w_C \) and \( m \) can be done in polynomial-time, we have \( \text{EC} \leq_p \text{Lang}_{\text{MIN-EC}} \). □

5.1.4 The Index Union Join Optimization Problem

The Index Union Join optimization problem (IUJO) asks for the best Index Union Join Plan: given a set of queries \( Q \) and a cost model \( CM \), find a set of indexes \( I \) and a set of unindexed queries \( Q_{UI} \) such that the cost given as the function \( \sum_{i \in IU} \text{IndexCost}_{CM}(\text{Type}_i)(i) \) is minimal, where \( IU = I \cup Q_{UI} \). The input language \( \text{L}_{IUJO} \) is here \( (Q, CM, IT, A) \in \text{AllPossibleQuerySets} \times \text{AllPossibleCostModels} \times \text{SetOfIndexTypesGivenCM} \times \text{SetOfAttributes} \) \(^{(3)}\). Additionally, we demand that the functions in \( CM \) are computable in polynomial-time.

Lemma 5.1.5. Index Union Join Optimization is in NPO

\(^{(3)}\) Note that further formalism is tedious and not the focus of the thesis.
**Proof.** Analogous to sub-section 5.1.3, we show that the three necessary conditions hold. Condition (1) is trivially met. Condition (2) is also met, because every plan has a size in \( O(|Q| \cdot |A| \cdot |IT|) \) and to verify that plan is a valid plan one has to check that each query belongs to exactly one index, which is in \( O(|Q|) \). Because we demanded that the index-cost functions in a cost model are in polynomial-time, condition (3) is also met.

\[ \square \]

**Theorem 5.1.6.** *Index Union Join Optimization is NP-hard.*

**Proof.** It is sufficient to show that \( \text{MIN-EC} \preceq_{p} \text{IUJO} \) holds.\(^{(4)}\) Let \( x = (W, C, w_{C}) \) be an element of the input language \( L_{\text{MIN-EC}} \). Let \( w_{i} \) denote the weight of a subset \( i \) of \( C \). We define now \( IT \) with only one type \( IT_{\text{MIN-EC}} = \{ \text{IndexType}_{\text{MIN-EC}} \} \) and the cost model \( CM_{\text{MIN-EC}}(w_{C}) \) with only one cost function \( \text{IndexCost}_{\text{MIN-EC}} \) as:

\[
\text{IndexCost}_{\text{MIN-EC}}(\text{IndexType}_{\text{MIN-EC}}(i)) = \begin{cases} 
  w_i & \text{if } i \in C \\
  \infty & \text{otherwise}
\end{cases}
\]

Now it holds for a given \( x \) that\(^{(5)}:\)

\[
\text{Opt}_{\text{MIN-EC}}(x) \leftrightarrow \text{Opt}_{\text{IUJO}}(W, CM_{\text{MIN-EC}}(w_{C}), IT_{\text{MIN-EC}}, \{\})
\]

All the input transformations can be done in polynomial time. \( \square \)

### 5.1.5 Search Space of the Index Union Join Optimization Problem

To define the search space we do the following consideration: One has to assign each query to at least one index or put it in \( Q_{\text{UI}} \). If each query fits in each index, then each query has \( |IT| \cdot |A| + 1 \) possible places in the plan. This is a search space of:

\[ (|IT| \cdot |A| + 1)^{|Q|} \]

In Crescando, \( |Q| \) has currently a maximum size of 4096. For a schema with thirty attributes and assuming there is one possible index type, the maximum search space would be: \((1 \cdot 30 + 1)^{4096} = 4.1 \times 10^{6108}\)

### 5.2 Related Work

Query plan optimization in a relational database is a well studied problem, first discussed in the System R paper [2]. A well known plan enumeration algorithm is based on a dynamic approach [21]. This dynamic programming search strategy generates \( O(3^{N}) \) possible plans for \( N \) table joins. Another interesting technique is a top-down approach with upper and lower cost bounds [25]. Yet, Crescando addresses a different optimization problem. There are no table joins, instead one can interpret a set of pending queries as a relation of predicates. In [3] this called

\(^{(4)}\) The property of NPO is here not needed

\(^{(5)}\) We omit the detail of transforming \( W \) into a query-set
the query-data join. The idea of a query-data join based on predicate indexes was first presented by Unterbrunner et al. in [29]. Because this paper was written in 2009, it is a relatively new problem. The idea of indexing predicates is inspired by publish-subscribe systems [8].

5.3 Optimizers

As illustrated in the last section, an extensive search is not feasible. Moreover, there is no intuitive way to split the problem into sub-problems where the local optimality property holds that is required for a dynamic programming approach. For example, the optimal plan for ten queries could be completely different from the optimal plan for the same ten queries and ten additional queries.

This section presents various optimizers, which were implemented as part of this thesis. The algorithms are described in pseudocode. The notation of an Index, describing a set of queries, is here reused. As described in the last chapter, Index denotes not an actual index used in the final plan. IndexSet IS is a set of indexes, and with IS.UI we denote the index with the type TypeUI of the set IS. As input, every optimizer gets a QuerySet, a CostModel and an IndexSet. Each optimizer returns a PlanModel from which the actual plan is built. PlanModel consist of a IndexSet PI, that describes a set of possible indexes, and a IndexSet AI, that describes a set of active indexes. Activating an index is moving it from PI to AI. Only indexes in AI are later built to form a real plan. Queries may be in AI or in PI but not in both. If they are in AI they are in exactly one index. Plans, in which not all queries are in AI, are called intermediate plans in contrast to complete plans. The function CM.Cost(PlanModel) sums up only the index cost from AI.

5.3.1 K-Greedy Distribute

This optimizer is a k-wise implementation of the already introduced GreedyOptimizer with an optimization called Distribute. When the old GreedyOptimizer builds an index, it inserts all queries into it that fit and are not already indexed. This means that, the sequence in which indexes are built matters heavily. Distribute means that in addition to moving all matching unindexed queries into the new index, queries that are already indexed are moved also to this index if the plan costs decrease. This corrects earlier mistakes to a certain degree.

The KGreedyDistribute optimizer is illustrated in algorithm 3. At each iteration, it builds $k^2$ new plans and chooses from them the k best plans. If all queries are in AI, a plan will generate no new plans. One can stop iterating if no new plans are generated and finally return the best found plan. The function $k\text{-min}_{\text{func}}$ returns a set of at most k elements, which are the smallest elements in the given set according to func. The function ActivateRefreshDistribute given in algorithm 4 activate an index, distributes queries and refreshes the values in PI.

---

(6) The same consideration can be done from the viewpoint of indexes
Algorithm 3: KGreedyDistributeOptimizer

Input: QuerySet QS; CostModel CM; IndexSet IS; int k;
PlanModel p; p.PI ← IS;
foreach Query q ∈ QS do // Initialize plan for 'minimum indexcost'
    foreach Index idx ∈ p.PI do // Add query to all not active indexes
        if q isIndexableBy idx then idx ← idx ∪ q;
    end
end
PlanModelSet CurrentP ← {p};
PlanModelSet NewP ← ∅;
repeat
    foreach PlanModel p ∈ CurrentP do // Build k * k new plans
        IndexSet KBest ← k-min CM.IndexCost(j) / |Q_j| {j ∈ p.PI};
        foreach Index idx ∈ KBest do
            PlanModel copy ← p;
            ActivateRefreshDistribute(QS, copy, idx);
            NewP ← NewP ∪ copy;
        end
    end
    if NewP ≠ ∅ then
        CurrentP ← k-min CM.Cost(p) / ∑_{i ∈ p.AI} |Q_i| {p ∈ NewP}; // Choose k best plans
    end
until NewP = ∅;
return min CM.Cost(p) {p ∈ CurrentP};

Algorithm 4: ActivateRefreshDistribute

Input: QuerySet QS; PlanModel p; Index newIndex;
foreach Index idx ∈ p.PI do
    idx ← ∅;
end
foreach Query q ∈ QS do // Query is already in an index; Distribute
    if q ∈ p.AI then
        Index oldIdx ← p.IndexOfQuery(q);
        Cost currentCost ← (CM.IndexCost(oldIdx) + CM.IndexCost(newIdx));
        oldIdx ← oldIdx \ q; newIndex ← newIndex ∪ q;
        Cost newCost ← (CM.IndexCost(oldIdx) + CM.IndexCost(newIdx));
        if newCost > currentCost then
            oldIdx ← oldIdx ∪ q; newIndex ← newIndex \ q;
        end
    else
        // Query not indexed
        if q isIndexableBy newIndex then // Query fit in new index; Add it
            newIndex ← newIndex ∪ q;
        else
            // Add query to all not active indexes
            foreach Index i ∈ p.PI do
                if q isIndexableBy i then i ← i ∪ q;
            end
        end
    end
end
p.PI ← p.PI \ newIndex; p.AI ← p.AI ∪ newIndex; // Activate Index
5.3 Optimizers

5.3.2 K-Backtrack Distribute

The KBacktrackDistribute optimizer is based upon the same ideas as the previous optimizers. However, it follows a depth-first search practice and uses recursive calls. Algorithm 5 shows the initial call and algorithm 6 is then the recursive function. Unlike a normal backtrack algorithm, which examines the whole search space, each call does at most \( k \) recursive calls. \( k \) can be computed at each iteration depending on the current recursive level. This allows to do more calls at the beginning, where it is more important, and later call a faster SimpleGreedyOptimize algorithm\(^7\) until the plan is complete. Reducing \( k \) is necessary; if \( k \) is not reduced, \( O(k^{RecursionDepth}) \) plans have to be tested, whereas the KGreedyDistribute optimizer does not need it because it tests only \( O(k^2) \) plans.

The backtrack method allows to use bounds. Observe that the upper bound \( uBound \) is only computed on a complete plan\(^8\). If now an intermediate plan has already a higher cost then this bound, the algorithm do not have to investigate this branch any more. A better bound check can be made, if the cost model can give a conservative estimate of how much it costs at least to get a plan completed (see LeastNeededCost).

Finally, the line IdxRemove in the algorithm 6 prevents the algorithm to build similar plans. Without this line, the optimizer likely builds plans where only the sequence of the indexes differ\(^9\). If one assume that the sequence of building indexes does not matter because of the ‘Distribute’, than these similar plans should be avoided.

Algorithm 5: KBacktrackDistributeOptimizer

Input: QuerySet QS; CostModel CM; IndexSet IS;
PlanModel p; p.PI ← IS;
foreach Query q ∈ QS do // Initialize plan for ‘minimum indexcost’
    foreach Index idx ∈ p.PI do // Add query to all not active indexes
        if q isIndexableBy idx then idx ← idx ∪ q;
    end
end
KBacktrackDistribute(p, CM.IndexCost(p.PI.UI), 0, CM);
return p;

Function computeK

There were three variants of computing \( k \) implemented. To ensure that no method does too many recursive calls, a safeguard is implemented with noRecursiveCalls > MAX-CALLS where MAX-CALLS is a constant. computeK has an initial value \( k_0 \) and a parameter recLvl, which denotes the current recursion depth.

\(^7\) Does the same as GreedyOptimizer but with ‘Distribute’ and bound checking
\(^8\) As initial value the cost is the same as if all queries are not indexed
\(^9\) KGreedyDistribute has this problem too and there it is not clear how to prevent the optimizer from building such solutions
• **Variant 0**: just returns the initial value. This results in $O(k_0^{RD})$ recursive calls where RD denotes the final recursion depth, which is the same as the final number of active indexes.

• **Variant 1**: subtracts from $k_0$ the current recursion depth recLvl. When recLvl $\geq k_0$, it returns one. This variant inspects then $O(k_0!)$ plans.

• **Variant 2**: returns $k_0$ as long as recLvl < $k_0$. If recLvl $\geq k_0$ it returns $2 \cdot k_0 - \text{recLvl}$. If recLvl $\geq 2 \cdot k_0$ it returns 1. For example the sequence of a k for $k_0 = 3$ would look like 3, 3, 3, 2, 1, 1, .... This results in $O(k_0 \cdot (k_0 - 1)!)$ recursive calls.

---

**Algorithm 6: KBacktrackDistribute**

```
InputOutput: PlanModel p;
Input: Cost uBound; int recLvl; CostModel CM;

int k ← ComputeK(recLvl);
if (k = 1) \lor (noRecursiveCalls > MAX-CALLS) then
  return SimplyGreedyOptimize(p, CM, uBound);
end

noRecursiveCalls ← noRecursiveCalls + 1;
IndexSet KBest ← k-min_{CM.IndexCost(i)/|Q_i|} \{j ∈ p.PI\};

if KBest = ∅ then return true; // All operations are indexed
if CM.LeastNeededCost(p) > uBound then
  return false; // Not possible to beat current bound
end

PlanModel bestP ← EmptyPlan;
foreach Index idx ∈ KBest do
  if CM.Cost(p) + CM.IndexCost(idx) < uBound then
    // Intermediate Cost is not higher than uBound
    if Type_{idx} = Type_{UI} then // All indexed, no recursive call
      bestP ← p; ActivateRefreshDistribute(QS, bestP, idx);
      uBound ← CM.Cost(bestP);
    else
      PlanModel copy ← p; ActivateRefreshDistribute(QS, copy, idx);
      Boolean success ← KBacktrackDistribute(copy, uBound, recLvl + 1, CM);
      if success then
        bestP ← copy; uBound ← CM.Cost(bestP);
      end
      p.PI ← p.PI \ idx;
    end
  end
end

IdxRemove

if bestP = EmptyPlan then return false;
p ← bestP;
return true;
```

---

### 5.3.3 Index Merger

The IndexMerger optimizer is based on two ideas. If there is no index lookup cost, then the best solution is the one, where each query is in the index in which it has the lowest probability to be evaluated. This is the initial solution. Because of lookup costs, the number of active indexes has to be decreased. Here the algorithm uses the second idea of merging an index with all other active indexes. An index
5.3 Optimizers

is chosen from a complete plan. This index is then removed and the queries of this
index are distributed between the other indexes. If the cost of the plan does not
shrink, the modification is reverted. This is done with each index, but as one can
easily see, the sequence in which indexes are merged matters (see line IdxChoose).
The current implementation uses a greedy method: It takes the least performing
index defined as \( \max_{j \in p.AI} \frac{CM.\text{IndexCost}(j)}{|Q_j|} \).

Note that the merging process is working on a complete plan and does never
increase the plan cost. So this can be also applied on any other plan derived from
another optimizer algorithm. The function \( \text{IndexForAttribute} \) takes an at-
ttribute \( a \) and returns an index \( idx \) from an IndexSet such that \( a_{idx} = a \) and that
a given query \( q \) fits in \( idx \) (e.g. if \( q \) has a range predicate on \( a \), a range index is
needed).

\[
\text{Algorithm 7: IndexMerger}
\]

**Input**: QuerySet \( QS \); CostModel \( CM \); IndexSet \( IS \); AttributeSet \( A \);

// Build initial solution
PlanModel \( p \); p.PI ← IS;

foreach Query \( q \in QS \) do
    Attribute \( a \leftarrow \min_{selectivity(a,q)} \{|a \in A\}; \)
    Index \( idx \leftarrow \text{IndexForAttribute}(p.PI, a, q); \)
    \( idx \leftarrow idx \cup q; \)
end

foreach Index \( i \in p.PI \) do
    if \( i \neq \emptyset \) then
        \( p.PI \leftarrow p.PI \setminus i; p.AI \leftarrow p.AI \cup i; \)
    end
end

// Try to merge active indexes
IdxChoose

foreach Index \( idx \in (p.AI \setminus p.AI.UI) \) do
    Cost currentPlanCost ← \( CM.\text{Cost}(p); \)
    PlanModel backup ← \( p; \)
    p.AI ← p.AI \setminus idx; p.PI ← p.PI \cup idx;
    foreach Query \( q \in idx \) do
        Index bestIdx ← p.AI.UI;
        \( idx \leftarrow idx \setminus q; bestIdx \leftarrow bestIdx \cup q; \)
        foreach Index newIdx ∈ p.AI do
            if q isIndexableBy newIdx then
                Cost currentCost, Cost newCost;
                currentCost ← \( (CM.\text{IndexCost}(bestIdx) + CM.\text{IndexCost}(newIdx)); \)
                bestIdx ← bestIdx \setminus q; newIdx ← newIdx \cup q;
                newCost ← \( (CM.\text{IndexCost}(bestIdx) + CM.\text{IndexCost}(newIdx)); \)
                if newCost > currentCost then
                    bestIdx ← bestIdx \cup q; newIdx ← newIdx \setminus q;
                else
                    bestIdx ← newIdx;
                end
            end
        end
    end

if \( CM.\text{Cost}(p) > currentPlanCost \) then \( p \leftarrow \) backup;
end

return \( p; \)

\[ (10) \text{Because the index cost is divided by the number of queries, smaller indexes are likely to have a higher score} \]
5.4 Evaluation

This section partially evaluates the plan enumeration algorithms in a well-behaved setting. The tests are designed in such a way that the GreedyOptimizer would build a bad plan. They are using a simple table schema with two integer attributes A and B. A predicate on any attribute could have a moderate selectivity m or a high selectivity h. There are only equality predicates used. All test runs use the new index-aware cost model and the CountMin sketch for selectivity estimation. Because of the small number of distinct values and the configured minimum size of a sketch, the selectivity estimation error is negligible. As k parameter, KGreedyDistribute and KBacktrackDistribute are using booth three and the later uses the variant (1) to compute the actual k. The optimizers have only to consider four possible indexes: two different attributes and for both there are two possible index types. The benchmark enqueues many queries at once such that the optimizer builds plans on the maximum number of concurrent queries, which is 4096.

5.4.1 Index Quantity Test

This test is designed such that the GreedyOptimizer builds too few indexes. The workload has two kinds of queries that are both equally likely: Q_{hA} has a predicate on attribute A with high selectivity and Q_{mA,B} has a predicate on A with moderate selectivity and a predicate on B with a high selectivity. In regard to the evaluation cost, the best plan would be to build an index on B with all queries from Q_{mA,B} and an index on A with the queries Q_{hA}. However, the GreedyOptimizer builds first an index on A and then puts all queries in that index. The test results have shown that all the new algorithms build the good solution.

Figure 5.1: Execution time and estimated cost of created plans
5.4 Evaluation

The figure 5.1 shows the execution time and the estimated cost for one scan, meaning this is for 4096 queries. One can see that the good solution does not make a big difference in execution time and plan costs. This is mainly because the schema has only a few attributes, which makes the evaluation of a query not very expensive.

In the case of the KBacktrackDistribute algorithm, the logs from the tests reveal something more: The first plan that it considers is the one with an MultiHashMap index on $A$. This removes also the MultiHashMap index on $A$ from the set of possible indexes for further plan creations. Then it considers a plan with an index on $B$ and then indexes the rest of the queries with an OneDimRTree on $A$. One can see here that the index pruning may not be such a good idea. If there would be not an other possible index type for $A$ the KBacktrackDistribute would build the same bad plan as the GreedyOptimizer, now it takes ‘only’ a suboptimal index type.

5.4.2 Index Quality Test

In this test, all algorithms are building two indexes. But the GreedyOptimizer assigns some of the queries in the wrong index. The workload consists of three different kinds of queries. There are queries $Q_{hA}$ which have a predicate on $A$ with a high selectivity. They are slightly more probable the the other two kinds of queries. $Q_{mB}$ queries have a predicate on $B$ with a moderate selectivity and $Q_{mAhB}$ have a predicate on each attribute with a moderate selectivity on $A$ and a high selectivity on $B$. The GreedyOptimizer builds an index on $A$ and puts then queries of the kind $Q_{hA}$ and $Q_{mAhB}$ in it and then it builds an index on $B$ with $Q_{mB}$. In terms of evaluation costs, it would be better to put the queries of the kind $Q_{mAhB}$ in the index on $B$.

![Figure 5.2: Execution time and estimated cost of created plans](image)
Figure 5.2 shows the results. Again this is for a scan with 4096 queries and the execution time difference is not very big because of the relative small query evaluation cost. However, The cost estimations show that all the new plan enumeration algorithms build the good plan.

5.4.3 Discussion

It is difficult to design tests where the new optimizers fail. In order that a KGreedy-Distribute or a KBacktrackDistribute optimizer builds too few indexes, one needs a schema of at least \( k + 1 \) different attributes and the workload needs to be rather exotic. Additionally, it is not clear how an ‘Index Quality’ test has to look like in order that algorithms with the ‘Distribute’ feature fail. This holds also for the IndexMerger. A workload in which the IndexMerger fails, is one in which an index that would be good is never considered as the initial solution does not contain that index. But such a workload seems not to be realistic.

The index pruning in the KBacktrackDistribute optimizer seems to be not such a good idea. The assumption that the order in which indexes are built does not matter, seems to be too strong. As the evaluation showed, in some cases a good index is not considered at all. An index \( I \) is built and all queries that fit into it are assigned in this index. Some of those queries would also fit in an index \( J \). But \( J \) is now not built because it does not pay off anymore. But if \( J \) was considered before \( I \) was built, \( J \) would pay off, or even more, \( I \) is not worthwhile.
Chapter 6

Overall Evaluation

This chapter evaluates the presented enhancements in terms of performance in a realistic setup.

6.1 Setup

To evaluate the system, we utilized a client application that links against the C-interface of the schema dependent library. This benchmark application is mainly written in Java and was developed as a benchmark in [10].

6.1.1 Test Platform

All the tests (also the previous ones) were run on a machine with 4 quad-core AMD Opteron 8354 (‘Barcelona’) processors and 32 GB of DDR2 667 RAM. Each core was run with a 2.2 GHz clock frequency, had 64 KB + 64 KB (data + instruction) L1 cache and 512 KB L2 cache. The machine was running a 64-bit Linux SMP kernel, version 2.6.32-25.

6.1.2 Amadeus Schema & Data Distribution

All benchmarks used the Amadeus Itinerary schema. Amadeus is a world-leading service provider for managing travel-related bookings. The Itinerary schema originates form a materialized view of their electronic marketplace. The schema consists of 47 attributes (see appendix 7.1 for the definition). Each test creates a table according to this schema and then loads data in it. The data distributions of each attribute are taken from the real Amadeus Itinerary table, in which most of the attributes have a more or less skewed distribution. This real-world data distribution serves as a good test scenario for the selectivity estimation.

6.1.3 Amadeus Workload

This query workload is based on the real-world workload on the Amadeus Itinerary table. It is an index-aware workload, meaning that most of the queries share the
same equality-predicate attributes. The baseline optimizer builds on this workload one up to two MultiHashMap indexes. A different execution plan does not promise a better performance, the baseline optimizer builds the best plan for this workload.

### 6.1.4 Synthetic Workload

As motivated in the introduction, we are interested in increasing the performance of Crescendo for unpredictable workloads. The synthetic workload is a random workload that has in the average 9 predicate attributes per query. The probability distribution for the predicate attributes is skewed and computed by the algorithm 8. The algorithm is executed with $N = 47$, which is the number of total attributes in the Amadeus Itinerary schema, and with $D = 9$, that is the already mentioned averaged number of predicate attributes. $s$ is the zipf distribution parameter. If not otherwise denoted, the tests used $s = 1$. Figure 6.1.4(1) illustrates the attribute distributions for different $s$ values, and the one from the Amadeus workload.

#### Algorithm 8: Attribute Pick Algorithm from [29]

<table>
<thead>
<tr>
<th>Input: $N, D, s$</th>
<th>Output: $P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Z \leftarrow \text{zipf}(s, N);$ // init $Z$ as zipf dist.</td>
<td></td>
</tr>
<tr>
<td>$V \sim \text{B}(N, 1/D);$ // $V$ acc. to binomial dist. $B$</td>
<td></td>
</tr>
<tr>
<td>for $v = 1$ to $V$ do</td>
<td></td>
</tr>
<tr>
<td>$a \sim Z;$ // get random $a$ according to $Z$</td>
<td></td>
</tr>
<tr>
<td>$p \leftarrow Z[a];$ // get prob. $p$ of $a$ acc. to $Z$</td>
<td></td>
</tr>
<tr>
<td>$Z[a] \leftarrow 0;$/ // remove $a$ from $Z$</td>
<td></td>
</tr>
<tr>
<td>$Z \leftarrow Z/(1-p);$ // re-normalize remaining $Z$</td>
<td></td>
</tr>
<tr>
<td>$P \leftarrow P \cup a;$ // add $a$ to result set $P$</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.1: Attribute frequency for vary $s$

### 6.1.5 Manually Corrected Cost Model

As we have seen in section 4.7, the cost-/executiontime ratio for a MultiHashMap index is noticeably higher than the one of the other two index types. For the evaluation, we implemented a cost model that computes the costs the same way as the index-aware cost model but reduces the estimate cost of an MultiHashMap index: $0.8 \cdot \text{CM.IndexCost(Type_{HM})}$, where $0.8 \approx \frac{\text{factor}_{JA} + \text{factor}_{RT}}{\text{factor}_{HM}}$. The factor values are taken from the figures in section 4.7. Because the factor values may differ in another setting, this cost model should not be used elsewhere. If not otherwise

---

(1) Data provided by [29]
specified, all tests in this section used this cost model because tests have shown that performance is in general slightly higher.

6.1.6 General Benchmark Settings
As stated in chapter 2, Crescando can run several scan threads. Because the query-execution plan is created within the scan thread, our changes to the system do not interfere with the performance in the presence of multiple scan threads. Hence, all the benchmarks are configured to use only one scan thread. Furthermore, the tests are using a read-only workload, as update queries are optimized the same way. The scan thread used a data segment of one GB size. At the start of a benchmark, data is loaded into the table up to a fill ratio of .75, so 75% of the one GB memory is used. Then, the benchmarks enqueue each second a constant number of read-queries. This number is higher than the system can handle. When Crescando’s query queue is full, additional queries are dismissed. The benchmark runs for fifteen minutes and counts the number of actually executed queries. The tests measure the peak throughput, which is the number of executed queries divided by the elapsed time. All the values in the figures of this chapter are average values of executing the a test four times, each time with an different data- and workload-generation seed.

6.2 Evaluation
We are in particular interested in the following questions:

- Does the improved selectivity estimation pay off? Which sketch is better?
- Does an improved cost model pay off?
- What is a good plan enumeration algorithm? What are good parameters for KGreedyDistribute and KBacktrackDistribute?
- Is there a novel plan enumeration algorithm that has in any setting an better performance then the baseline optimizer?
- How much performance did we gain?

Each of the following subsection presents benchmark results to answer those question.
6.2.1 Performance of the Novel Selectivity Estimation

The benchmark was run once with the old selectivity estimation and once with each sketch. Figure 6.2 show the results for various plan enumeration algorithms and the two workloads. \textit{IndexUnion} denotes the presented baseline optimizer. The plan enumeration algorithms are using the manually corrected cost model, except for the IndexUnion optimizer, which is not capable of using this model. ‘KGreedyDis-3’ is the KGreedyDistribute algorithm using $k = 3$ and ‘KBacktrackDis-1-3’ is the KBacktrackDistribute algorithm using also $k = 3$ and the variant (1) to compute the actual $k$ value (see 5.3.2).

In the case of the Amadeus workload, the usage of the sketches do not harm, but no configuration is noticeable faster than the baseline configuration. In the case of the synthetic workload, Crescando is considerable faster with the sketches. The FastAGMS sketch seem to perform slightly better. Note that the slow counting in case of the FastAGMS is in this benchmark not an issue, because the sketches are built at the startup time and during the measurement there are no sketch rebuilds\(^{(2)}\). The higher estimation costs of the FastAGMS sketch seem to be not an issue.

Summarizing, a selectivity estimation on a per-value level does pay off and CountMin and FastAGMS perform nearly the same in this benchmark. But CountMin is preferable to FastAGMS because of its faster behavior at construction shown in chapter 3.

(a) Amadeus

(b) Synthetic

Figure 6.2: Peak throughput for different selectivity estimators

\(^{(2)}\)Recall that Crescando refreshes selectivity estimates after a certain amount of data was changed and that the benchmarks do not have write queries.
6.2 Evaluation

6.2.2 Performance of the New Cost Model

This time, the selectivity estimates are fixed to the FastAGMS sketch and the cost models are varied. Again, the benchmarks were run with different optimizers and workloads. The figure 6.3 shows the results. One can see a very low performance of the KBacktrackDistribute algorithm with the baseline cost model in both workloads. It is not know from where this misbehavior originates. In the Amadeus workload the new cost models are not helpful. The worse performance cannot be explained only with the increased overhead of the new cost model.

Because the baseline cost model has no lookup costs, the IndexMerger does not merge any index. One can see that this is a good choice for the Amadeus workload. This means that the best solution is the initial solution of the IndexMerger: each query is in the index in which it has the highest selectivity. Cost models which have lookup costs forces the other optimizers to built less indexes. This is the reason for the poorer performance of the new cost models in the Amadeus workload.

However, in the synthetic workload the IndexMerger builds too many indexes with the baseline cost model. This shows how harmful too many indexes are. In the synthetic workload all the optimizers profit from the new cost model. For unpredictable workloads one can say that the new cost model pays off, and depending on the plan enumeration algorithm, it is even necessary. Also, a better initialized cost model would be a gain as the comparison of the index-aware with the manually corrected cost model shows.

Figure 6.3: Peak throughput for different cost models
6.2.3 Comparison of the Novel Plan Enumeration Algorithms

Here, benchmarks were run with the manually corrected cost model and FastACMS for various plan enumeration algorithms and their parameters. For the KGreedyDistribute optimizer, $k$ was varied between one and five. For the KBacktrackDistribute, $k$ was varied between one and four (last digit in the labels) and between the three $k$-computation functions described in section 5.3.2. The black vertical lines illustrate measurement fluctuations. They begin at the lowest measured throughput and end at the highest measured throughput of the four benchmark runs. The blue bars show the already known average peak-throughput, whereas the red ones show the time needed to ‘optimize’ a single query.

Figure 6.4: Peak throughput for different optimizers with the Amadeus workload

Figure 6.4 shows the results of the benchmarks on the Amadeus workload. Here, with an optimization time of $25\mu s$ per query, the system spends about 2% of the whole time with optimizing. It seems that increasing $k$ on the KGreedyDistribute optimizers does not result in better plans. Additionally, the optimization time increases. In the case of the backtrack algorithms, an higher $k$ or changing the $k$-computation function does not have a big influence on the optimization time. It is assumed, that this is because of the bounds in the algorithm. Together with the KBacktrackDis-0-2, the IndexMerger results in the fastest execution. Except for the KGreedyDistribute algorithms with an high $k$, the normal Greedy has the worst performance. But as the last subsection showed, the Greedy optimizer would be as fast as the IndexMerger with the baseline cost model in the Amadeus workload.

Figure 6.5 shows the peak throughput in the case of the synthetic workload.
Here, with an optimization time of 100µs per query, the system spends about 1% of the whole time with optimizing. In the case of KBacktrackDis-0-4, this is about 3%\(^{(3)}\). Surprisingly, this optimizer is still better than the Greedy optimizer and it is not much slower than other algorithms. Again the optimizers with a small k parameter are better, but in general there are only small speed differences. For the KBacktrackDistribute algorithm, the k-computation function (1) seems to be a good choice.

In average a plan has about 20 different indexes. Therefore KGreedyDis-5 considers about $5 \cdot 5 \cdot 20 = 400$ different plans, whereas the KBacktrackDis-2-4 considers about $4^4 \cdot 3 \cdot 2 \cdot 1 = 1536$ plans. But KGreedyDis-5 and KBacktrackDis-2-4 are needing about the same optimization time, which is only possible thanks to bounds where KBacktrackDistribute can prematurely discard plans.

In conclusion, the only important improvement for the plan enumeration algorithms is the ‘Distribute’ feature: given a set of indexes that will be built, the queries have an optimized distribution\(^{(4)}\). This is the speedup between the Greedy optimizer and the other optimizers in the synthetic workload. For the presented plan enumeration algorithms, it does not pay off to spend more time for optimizing. For the KGreedyDistribute and KBacktrackDistribute choosing $k = 2$ seems to be a good choice.

\(^{(3)}\) Note that KBacktrackDis-0-4 would consider much plans, but it is prematurely stopped by the safeguard described in 5.3.2
\(^{(4)}\) The IndexMerger has this property too
Varying Workload Diversity

To find an overall good optimizer, we have run the benchmarks with varying the s parameter of the synthetic workload. With varying s one can simulate different workload diversity levels. The figure 6.6 shows the peak throughput relative to the baseline optimizer. **All** the benchmarks used the FastAGMS sketch and except for the IndexUnion they use the manually corrected cost model. Note that with the baseline cost model, the Greedy optimizer would behave very likely to the IndexUnion optimizer\(^{(5)}\). The IndexMerger is clearly the best plan enumeration algorithm, because it behaves for any s as least as good as any of the other optimizers.

![Figure 6.6: Performance with different zipf parameters](image)

\(^{(5)}\) Actually, this was the design goal of the Greedy optimizer
6.2 Evaluation

6.2.4 Baseline vs. New Optimal Approach

The figure 6.7 shows benchmark results for the baseline optimizer with the old selectivity estimates and of the IndexMerger with the FastAGMS sketch and the manually corrected cost model. As one can see, there is no difference in the old Amadeus workload but in the synthetic workload the new IndexMerger is nearly twice as fast.

![Comparison of IndexUnion with IndexMerger](image)

Figure 6.7: Comparison of IndexUnion with IndexMerger
Chapter 7

Conclusion

As a result of this thesis, we have shown how a scan-based main-memory table can be improved in regard to multi-query optimization. The evaluation revealed that the resulting system is up to two times faster for unpredictable workloads. In case of index-aware workloads, it has the same performance as the old system.

From the three investigated optimization areas, one can state that the most important area is the selectivity estimation based on a per-value level. To gather and maintain the required meta-data, sketches are a fast solution having a small space usage. The CountMin sketch is faster at counting and estimating than the FastAGMS. However, the FastAGMS sketch revealed a better estimation accuracy for the real world setup in the overall evaluation.

The cost estimation can be improved even more, with an index-aware cost model. In a scan-based main-memory table, CPU costs can be modeled in terms of evaluation costs and index-dependent lookup costs.

Finally, a simple and fast greedy like approximation algorithm is in most cases sufficient, although the multi-query optimization problem is NP-hard. The most important enhancement in the plan enumeration algorithms is the ‘distribute’ feature.

In conclusion, the optimization problem is hard whereas a good approximated solution can be found relatively easy. The more important part is an accurate estimation of index-costs or costs of a whole plan.

7.1 Future Work

A scan-based relational main-memory table still offers many opportunities to optimize multi-query execution. Maybe not for a index-aware workload but certainly for unpredictable workloads.

To gather statistical data one could try also another approach, such as counter-based solution presented in section 3.2. The evaluation indicates that the performance is very sensitive to the selectivity estimates. One should consider also a solution based on a hash table like the naïve implementation presented in 3.4.4,
which showed already a fast behavior. Most of the Amadeus attributes have (much) less than 2500 distinct values. For such attributes, a hash table counter appears to be affordable in terms of space. For other attributes, a sketch still seems to be a good solution.

It is assumed that improving further the cost model promises the highest performance gain of the three presented optimization areas. The OneDimRTree lookup cost estimation should get a more accurate formula, probably a function that uses the query range sizes. But the most worthwhile enhancement would be the modeling of space costs. If an additional index would cause (more) cache-trashing, the index-cost function should return an higher estimated cost. Expectations are that especially the IndexMerger would profit from this improvement.

As already mentioned, a KGreedyIndexMerger could be a good optimizer too. But as we have seen a simple and fast optimizer is doing it already very well, making such an optimizer not necessarily useful. It would be interesting to know the theoretical accuracy of the approximated solutions generated by these optimizers. The results indicates that these solutions are already very near at the optimum. There are two other very simple changes to the plan enumeration algorithms which should be evaluated: apply the index merging to a plan derived from another optimizer, especially to one from a ‘Distribute’ optimizer because ‘Distribute’ may ruin the usefulness of an index and evaluate the KBacktrackDistribute optimizer without the index pruning.

Another optimization is an out-of-order query evaluation. To a certain degree, queries can be delayed to a later scan without violating consistency constrains. For example one can delay all unindexed queries to the next scan or it can be done in a large-scale manner: a big query set is divided into two sets in order to have less indexes in each plan, avoiding cache trashing. But the option of delaying queries increases the complexity of the optimization problem.

Adding new indexes with different performance properties could enhance the system too. The new cost model and optimizer architecture allows adding and index with a relative small effort.

The whole thesis is based on the Index Union Join Plan. Probably, one could enhance this plan too. For example some sort of multi-attribute bloom filter that can filter out records that for sure do not match any query. This appears to be not a good idea, because the more queries are in a plan, the less useful is such a filter. But maybe someone else has a better one.
### Amadeus Itinerary Schema

```sql
CREATE TABLE Itinerary (  
  provider VARCHAR(3),  
  productId UINT(16),  
  alphaSuffix CHAR,  
  dateInFirstLeg DATE,  
  dateIn DATE,  
  dateOut DATE,  
  cityFrom VARCHAR(3),  
  cityTo VARCHAR(3),  
  cancelEnvelope INT(32),  
  cancelInitiator VARCHAR(3),  
  rloc VARCHAR(6),  
  paxTattoo INT(32),  
  segmentTattoo INT(32),  
  purgeDate DATE,  
  office VARCHAR(9),  
  creationDate DATE,  
  modificationDate DATE,  
  nip INT(16),  
  unassigned INT(16),  
  pnrQualifier UINT(64),  
  paxQualifier UINT(64),  
  sgtQualifier UINT(64),  
  name VARCHAR(57),  
  firstname VARCHAR(56),  
  sex BOOLEAN,  
  cabin CHAR,  
  classOfService CHAR,  
  bookingStatus VARCHAR(2),  
  codeShareType VARCHAR(2),  
  bookingDate DATE,  
  subclass UINT(8),  
  posCrs VARCHAR(3),  
  posCountry VARCHAR(2),  
  cancelFlag CHAR,  
  mariage VARCHAR(3),  
  yieldValue INT(32),  
  rvIndicator CHAR,  
  rvValue INT(32),
)
<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>cnxNumber</td>
<td>INT(8),</td>
</tr>
<tr>
<td>did</td>
<td>VARCHAR(16),</td>
</tr>
<tr>
<td>iid</td>
<td>VARCHAR(16),</td>
</tr>
<tr>
<td>indexingVersion</td>
<td>INT(16),</td>
</tr>
<tr>
<td>sgtVendorFormat</td>
<td>VARCHAR(16),</td>
</tr>
<tr>
<td>sgtVendorValues</td>
<td>VARCHAR(16),</td>
</tr>
<tr>
<td>inboundCnxTime</td>
<td>INT(32),</td>
</tr>
<tr>
<td>inboundSgtTattoo</td>
<td>INT(32),</td>
</tr>
<tr>
<td>outboundCnxTime</td>
<td>INT(32),</td>
</tr>
<tr>
<td>outboundSgtTattoo</td>
<td>INT(32)</td>
</tr>
</tbody>
</table>
Bibliography


