Optimization of Boolean Expressions for Main Memory Database Systems

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

Fisnik Kastrati, M.Sc.
aus Pristina

Mannheim, 2017

Dekan:	Dr. Bernd Lübcke, Universität Mannheim
Referent:	Professor Dr. Guido Moerkotte, Universität Mannheim
Korreferent:	Professor Dr. Christian Becker, Universität Mannheim

Tag der mündlichen Prüfung: 29. Januar 2018

Acknowledgments

First and foremost, I would like to express my deepest appreciation to my advisor prof. Guido Moerkotte. His guidance and supervision has enabled me to grow as a researcher, as well as become a better computer scientist in general. His immense knowledge in the area of query optimisation and databases has proved invaluable to me throughout my entire PhD journey. A special thank you goes to my second advisor prof. Christian Becker for reviewing my thesis and serving in my thesis committee.

I would like to thank Simone Seeger for her meticulous proofreading of my draft papers. Im grateful to my colleagues Marius Eich and Magnus Müller for their feedback and comments.

I would also like to acknowledge University of Mannheim for the dissertation completion grant (Landesgraduiertenförderung – LGF) which had removed the financial burden during my thesis writing.

Last but not least, I would like to thank my wife for her love, support and constant encouragement. Many thanks go as well to my parents who were very supportive throughout my PhD.

Abstract

With the ubiquity of main memory databases which are increasingly replacing the old disk-oriented databases, relations are being stored in denormalized form in order to increase the query throughput, thus, the dominance of join operators in terms of costs is being replaced by the costs of evaluating selection predicates. Boolean expressions containing selection predicates connected both conjunctively and disjunctively have been thus far solved by rather simple heuristics which leaves a large optimization potential unharvested. To exacerbate the matter, such heuristics rely on the independent predicate selectivity assumption which typically does not hold, and the constant predicate costs assumption which in terms of main memory database systems does not hold either. In this thesis we tackle the problem of optimizing Boolean expressions by not relying on the independence assumption nor the constant predicate costs assumption. We present optimization algorithms for queries containing both conjunctively and disjunctively connected predicates together with a cost model which precisely captures CPU architectural characteristics such as branch misprediction. Our optimization algorithms achieve the optimum in terms of plan quality, thus, they harvest the entire optimization potential inherent in Boolean expressions.

Zusammenfassung

Die sich wandelnde Hardwarelandschaft führt dazu, dass Hauptspeicher-Datenbanksysteme die klassischen disk-basierten Systeme zunehmend verdrängen. Hauptspeicher-Datenbanksysteme speichern Relationen in denormalisierter Form um den Durchsatz an Anfragen zu erhöhen. In Folge dessen übernehmen Selektionsprädikate die Rolle von Joins als entscheidenden Kostenfaktor. Konjunktiv sowie disjunktiv verknüpfte Boolesche Ausdrücke, welche Selektionsprädikate enthalten, wurden bisher von simplen Heuristiken optimiert. Dies hat viel Spielraum für Optimierungen gelassen. Die kritischen Schwachstellen dieser Heuristiken sind, dass sie sowohl annehmen, dass die Selektivtäten Unabhängigkeit voneinander sind, welches im Allgemeinen nicht gilt, sowie, dass die Evaluationskosten von Prädikaten unempfindlich gegenüber ihrer Selektivität sind. Letzteres spiegelt insbesondere in Hauptspeicher-Datenbanksystemen die Realität nicht wider. Diese Arbeit behandelt die Optimierung von Booleschen Ausdrücken ohne diese Annahmen. Wir zeigen Optimierungsalgorithmen für Anfragen, die sowohl konjunktiv als auch disjunktiv verknüpfte Prädikate enthalten. Darüber hinaus präsentieren wir ein dafür angepasstes Kostenmodell, welches Charakteristiken der Prozessorarchitektur, wie *branch misprediction*, präzise modelliert. Unsere Optimierungsalgorithmen sind optimal in Bezug auf die Planqualität; sie schöpfen das gesamte Optimierungspotential Boolescher Ausdrücke aus.

Contents

List of Figures

List of Tables

1. Introduction

Optimization of Boolean expressions in database system is a very challenging problem, which prompts for carefully designed optimization algorithms in order to harvest the large optimization potential inherent in them. With the advent of main memory databases, optimization of Boolean expressions becomes an even more challenging task requiring optimization algorithms which take into account hardware characteristics such as CPU branch misprediction. In the recent years, the memory price has drastically decreased and at the same time its size has drastically increased. For instance, in early 80s the cost per MB of main memory was around 6400 USD, whereas at the time of this thesis writing it is 0.0059 USD[1]. Servers with terabytes of main memory have now become affordable thus prompting a shift from disk oriented database systems to main memory oriented database systems. Consequently, this thesis focuses on optimization of Boolean expressions for main memory databases.

In the first part of this thesis, we consider the problem of optimizing Boolean expressions composed of predicates connected conjunctively. We present an efficient optimization algorithm for this class of queries which relies on dynamic programming and generates the solutions in a bottom-up fashion.

Boolean expressions containing predicates connected conjunctively and disjunctively are then the topic of the second and the third part of this thesis. We initially present an efficient heuristic optimization algorithm for disjunctive predicates which leverages bypass processing. Although the heuristic algorithm is superior to the existing heuristics in the literature, it does not attain the optimum in terms of plan quality. We can, however, achieve the optimum by means of the optimization algorithm presented in the third part of this thesis, which optimizes Boolean expressions in a top-down fashion. Top-down algorithms in contrast to bottom-up algorithm have the advantage of employing search strategies like branch-and-bound pruning in order to reduce the search space. Besides the branch-and-bound pruning search strategy, our top-down optimization algorithms make a use of the Boolean difference calculus in order to derive tighter upper bounds and this way prune even more aggressively the search space.

Optimization algorithms found in the literature—and commonly used in commercial relational database systems—rely on at least two assumptions: (1) predicate selectivities are assumed to be independent, and (2) predicate costs are assumed to be constant. Since both of these two assumptions typically do not hold, optimization algorithms presented in this thesis do not rely on any of them. Since we do not rely on the independence assumption, in the fourth part of this thesis we present a very efficient sampling method, which can be used to

[1]The prices were taken from http://www.jcmit.net/memoryprice.htm

gather predicate selectivities for all the subsets of predicates given in a query. To that end, optimization algorithms presented in this thesis work take into consideration CPU architectural characteristics such as branch misprediction penalty, as well as common subexpression elimination when present in Boolean expressions.

Since the cost model plays a principal role in query optimization, we present a cost model which very precisely models hardware characteristics such as branch misprediction penalty as well as cache misses. The cost model presented in this thesis is novel in that it shows a direct relationship between the error in the cost functions and the plan quality.

The rest of the thesis is organized as follows. In Chapter 2 we presented the preliminaries required to understand the subsequent chapters. Chapter 3 presents the system prototype used in this work. The cost model used in our optimization algorithms is presented in Chapter 4 together with the approximation framework which is used to obtain the parameters for our cost functions. The optimization algorithm for conjunctive Boolean expressions is the topic of Chapter 5. A heuristic for Boolean expressions containing both conjunctive and disjunctive predicates is presented in Chapter 6. Chapter 7 presents a top-down optimization algorithm for Boolean expressions which attains the optimum in terms of plan quality. Chapter 8 presents a sampling method which efficiently gathers predicate selectivities. Furthermore, in Chapter 8 we show how the predicate selectivities gathered by our sampling method can be used when optimizing queries containing both conjunctive and disjunctive predicates. Finally, Chapter 9 concludes the thesis.

2. Preliminaries

In this chapter we present the preliminaries required to understand the work in this thesis.

Initially we give a brief introduction to the algebra, followed by some background information about the computer architecture. We conclude this chapter by presenting details on the two major materialization strategies used in main memory database systems.

2.1. Relational Model

The relational model was first introduced in 1969 by Edgar F. Codd, and since then it became the de facto standard for data representation in the database community. The relational model is quite pragmatic due to its fundamental building block: *mathematical relation*. The roots of the relational model come from set theory and first-order predicate logic.

In the relational model, a database is represented as a set of relations. A relation consists of set *tuples* where each tuple is composed of a number of <attribute, value> pairs. Tuples in a relation represent facts about some entity, or relationships. Relations can be informally thought of as two-dimensional *tables* consisting of *rows* and *columns*. Rows represent tuples, whereas columns represent attribute values drawn from a finite domain.

Movie	Year	IMDb rating
The Godfather	1972	9.2
The Dark Knight	2008	8.9
Pulp Fiction	1994	8.9
A Beautiful Mind	2001	8.2

Table 2.1.: Movies relation

An example relation about movies has been depicted in Table 2.1. This relation describes the movie title, the year when the movie first appeared, and its IMDb[1] rating.

The data-manipulation part of the relational model is defined in *relational algebra*. The relational algebra operators are divided into two groups. The first group include set operators coming from mathematical set theory. Such operators are set union ($\cup$), set difference ($\cap$), set intersection ($\setminus$) and cross product ($\times$). Since in relational model relations are defined as a set of tuples, the above enumerated set operators are applicable. If, however, duplicates are

[1] http://www.imdb.com/

to be considered, then we denote with ($\uplus$) the union operator without duplicate elimination.

$$e_i \cup e_j := \{t | t \in e_i \vee t \in e_j\}$$
$$e_i \cap e_j := \{t | t \in e_i \wedge t \in e_j\}$$
$$e_i \setminus e_j := \{t | t \in e_i \wedge t \notin e_j\}$$

The second group consists of the operators which were developed for relational databases. In the second group belong operators such as selection (σ), projection (π), join ($\bowtie$) among others.

The selection operator filters out all the tuples that do not satisfy the predicate p:

$$\sigma_p(e) := \{\, t \,|\, t \in e \wedge p(e)\}.$$

We make no restriction on the predicate p, it can include method calls, nested expressions, etc. Further, if the input of the selection does not contain duplicate values, the output is duplicate-free too.

The projection operator π can be used to remove attributes

$$\pi_A(e) := \{a_1 : x.a_1, \ldots, a_n : x.a_n \,|\, x \in e\}$$

whereas the operator which is used to create (compute) new attributes is the *map operator* χ [4, 44]: $\chi_{A_1:e'_1,\ldots,A_k:e'_k}(e)$. The map operator extends an input tuple by a new attribute A, whose value is calculated via an arbitrary expression e':

$$\chi_{A:e'}(e) := \{t \circ [A : v] \,|\, t \in e, v = e'(t)\},$$

where $\circ$ denotes the tuple concatenation operator. We generalize the map operator for many attributes as follows:

$$\chi_{A_1:e'_1,\ldots,A_k:e'_k}(e) := \chi_{A_k:e'_k}(\ldots \chi_{A_1:e'_1}(e)\ldots).$$

In relational algebra there exists many variants of join operators. Five of them are rather standard and encountered often in the literature. These are join, semijoin, antijoin, left outerjoin, and full outerjoin. We will only give the definition for the cross product and the regular join operator:

$$e_i \times e_j := \{x \circ y \,|\, x \in e_i \wedge y \in e_j\}$$
$$e_i \bowtie_p e_j := \{x \circ y \,|\, x \in e_i \wedge y \in e_j \wedge p(x, y)\},$$

the definitions for the rest of join operators can be found in [46].

The input to relational algebra operators are instances of relations, and the result of algebra operators are new relations. The newly produced relations can be an input to other operators and this way one can flexibly combine a sequence of relational algebra operations in a *relational algebra expression.* To this end, relational algebra expressions enable users to specify their information retrieval requests. As an example, lets assume that we are interested in finding all the movie names that appeared in the time period between 1990 and 2005 from the

movies relation depicted in Figure 2.5. This retrieval request can be expressed in relational algebra as follows:

$$\pi_{movie}(\sigma_{year \geq 1990 \wedge year \leq 2005}(Movies)).$$

Relational algebra is less expressive than conventional programming languages such as C++, Java, etc. That is, there are computations possible in programming languages which cannot be performed in relational algebra. However, limitations in its expressive power make relational algebra easier to write queries in, and further, they allow the optimizer to generate a highly optimized code.

2.2. Computer Architecture

In the following subsections, we present computer architectural details which are of relevance to understanding the material presented later. The material touching the hardware covers only background information for readers not familiar with the modern computer architecture and is by no means exhaustive. Readers interested in more details about the computer architecture are referred to the excellent book by David A. Patterson and John LeRoy Hennessy – "Computer Architecture: A Quantitative Approach" [27].

2.2.1. Instruction Pipelining in Modern Processors

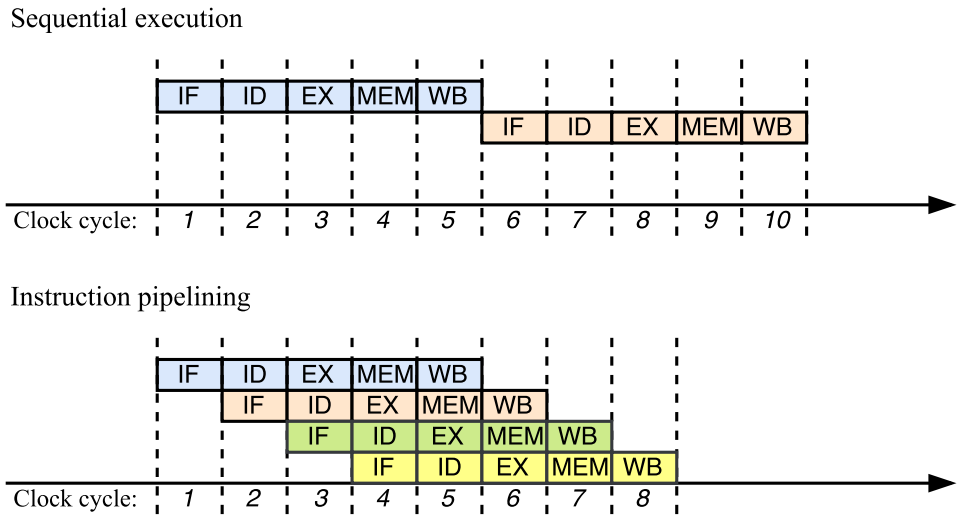


Figure 2.1.: Illustration of sequential execution and instruction pipelining in a five stage RISC processor

Execution in a processor is typically broken into a number of stages, where specialized processor units execute each stage. In the most basic computing model, the processor executes at most one instruction per clock cycle, this way only one execution unit is active at any clock cycle while other execution units

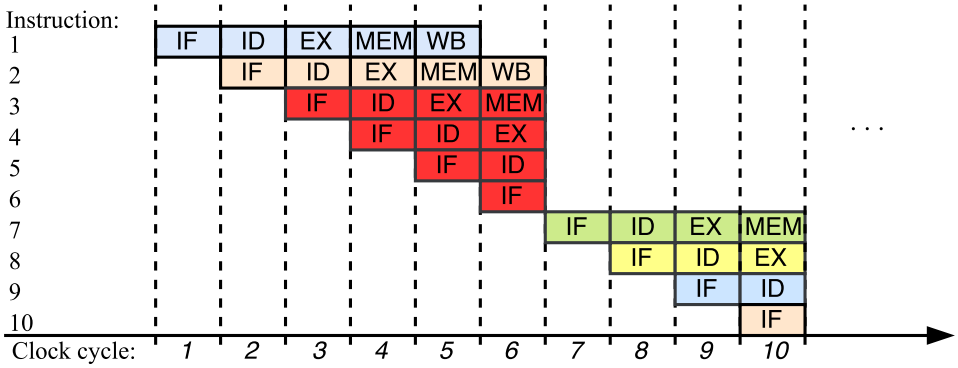


Figure 2.2.: Illustration of a pipeline stall

remain idle. This model is known as the *sequential execution model*. Modern processors, however, allow for more than a single instruction to be executed concurrently at any given clock cycle in order to increase the throughput. This computation model is known as *instruction pipelining*, or *instruction-level parallelism*.

Execution stages in a processor typically follow the pattern: instruction fetch (IF), instruction decode (ID), instruction execute (EX), memory access (MEM), and write back (WB), i.e., save the results in a register (if necessary). The MEM stage is applicable only when the instruction needs to access the data memory. The difference between a pipelined execution model and the sequential one has been illustrated in Figure 2.1. Note that modern CPUs have more stages than the ones shown in this illustration, e.g., Intel Haswell processor has a 14-stage pipeline and on the other extreme, Intel Prescott processor has a 31-stage pipeline.

2.2.2. Branch Misprediction

For a program without conditional statements, the code is simply a sequence of instructions allowing for a pipelined execution as shown in the bottom part of Figure 2.1. If the program, however, contains conditional statements, CPU tries to predict the outcome of the predicate and load into the pipeline instructions corresponding to the predicted execution path. This happens as the CPU cannot support simultaneously all the possible paths in pipelined execution, therefore it has to guess the execution path. If the guessed execution path turns out to be wrong, then several instruction in the pipeline have to be flushed, thus causing a *pipeline stall*.

Figure 2.2 illustrates the branch misprediction and the resulting pipeline stall. Assume that instruction 2 in the illustration is a conditional statement, and hence at clock cycle 3, the processor speculatively fetches instruction 3 and as a result, it executes the wrong sequence of the instructions 4,5, and 6. When instruction 2 is completely executed at clock cycle 6, the CPU detects that the sequence of the instructions 3-6 are wrongly executed, therefore it flushes the pipeline and in clock 7 starts loading the correct sequence of the instructions

7-10. When the instruction 7 is being fetched (IF), there is no overlapping with earlier instructions due to the pipeline being flushed, and therefore CPU resources are wasted. That is, the branch misprediction has delayed (stalled) the execution of the instruction 7 by 4 cycles, whereas the normal delay between two instructions is only 1 cycle, e.g., see the execution of the instructions 1 and 2 in Figure 2.2.

In real processors, delays due to control hazards like branch misprediction are much longer as there the number of pipeline stages is larger. In Intel i7-4770 Haswell processor the branch misprediction penalty is quite severe—it costs 18-20 cycles[2], which makes the optimization of this penalty critical. The optimization algorithms presented in Chapter 5, Chapter 6 and Chapter 7 judicially optimize this penalty in the context of main memory databases.

Branching and Non-branching Conditions

As mentioned in the previous section, the presence of conditional statements in a program can cause control-hazards due to branch misprediction(s). A conditional statement can be composed of multiple predicates connected conjunctively. Consider the following simple expression containing a conjunction $p_1 \wedge p_2$ of predicates. This conditional statement in programming languages like C/C++ can be evaluated by expressions either of the form p_1`&&`p_2 or of the form p_1`&`p_2. The evaluation of `&` is performed by first evaluating both its arguments. Then, the *logical and* ($\wedge$) is calculated by a *bitwise and* operation. The expression p_1`&&`p_2 is evaluated by first evaluating p_1. If p_1 evaluates to false, this is the result. If p_1 evaluates to true, then and only then p_2 is evaluated. The result of this evaluation is the result of the whole expression. To that end, the evaluation of the expression p_1`&&`p_2 includes a conditional branch, which introduces a possibility for branch misprediction. The evaluation of the expression p_1`&`p_2 does not include a conditional branch, although after its evaluation there might be one.

To better understand the branching AND (`&&`) and non-branching AND (`&`) logical connections, let us consider the following simple C/C++ code snippet:

```
bool branchingAnd(int p1, int p2) {
  if(p1 > c1 && p2 < c2) {
    return true;
  }
  return false;
}

bool nonbranchingAnd(int p1, int p2) {
  if(p1 > c1 & p2 < c2) {
    return true;
  }
  return false;
}
```

[2] http://www.7-cpu.com/cpu/Haswell.html

The first function `branchingAnd(int p1, int p2)` contains an if condition with the branching AND (`&&`) connection, whereas the second function contains the same condition, however, with a non-branching AND (`&`) connection. In the following, we show the assembly code generated when compiling these two functions using Intel's C++ compiler (`icpc` version 16.0.3):

```
1  branchingAnd(int , int):
2          cmp       edi, DWORD PTR c1
3          jle       ..B1.4
4          cmp       esi, DWORD PTR c2
5          jge       ..B1.4
6          mov       eax, 1
7          ret
8  ..B1.4:
9          xor       eax, eax
10         ret
11
12 nonbranchingAnd(int , int):
13         xor       eax, eax
14         mov       r8d, 1
15         xor       edx, edx
16         cmp       edi, DWORD PTR c1
17         cmovg     edx, r8d
18         xor       ecx, ecx
19         cmp       esi, DWORD PTR c2
20         cmovl     ecx, r8d
21         test      edx, ecx
22         cmovne    eax, r8d
23         ret
```

For the `branchingAnd(int p1, int p2)` function, the compiler has generated a conditional jump to location `..B1.4` if the condition $p1 > c1$ is not satisfied, as shown in line 3. In such case, the second condition is bypassed altogether and the function returns a false value. In the generated code, however, there is also a second conditional jump shown in line 5. The second jump corresponds to the second condition ($p2 < c2$), and is taken only if the first condition succeeds, but the second one fails.

For the `nonbranchingAnd(int p1, int p2)` function, there are no such conditional jumps in the generated assembly code. Both conditions are evaluated, and their results are stored in the registers `edx`, and `ecx`. These two registers will hold binary values (1 or 0) depending on the outcome of the respective conditions. In line 21, however, there is a bitwise test instruction, which performs a bitwise-AND over the registers `edx`, and `esx`. If the bitwise test yields 1, the C/C++ code inside the if statement will be executed, otherwise the code control will return to the point outside the if statement, that is, our function will return false. To this end, depending on the predicate selectivities, one should carefully choose either evaluation method in order to minimize the branch misprediction penalty. We present in Chapter 5 an optimization algorithm for conjunctive predicate, which judiciously chooses either logical connection (`&&` or `&`) depending on predicate selectivities in order to minimize the branch misprediction penalty.

Memory	Size	Latency
registers	< 1 KB	1-2 cycles
L1 cache	64 KB	4 cycles
L2 cache	256 KB	12 cycles
L3 cache	8 MB	36 cycles
main memory	GB to TB	50 - 200 cycles

Table 2.2.: Memory access times in a Intel i7-4770 Haswell processor

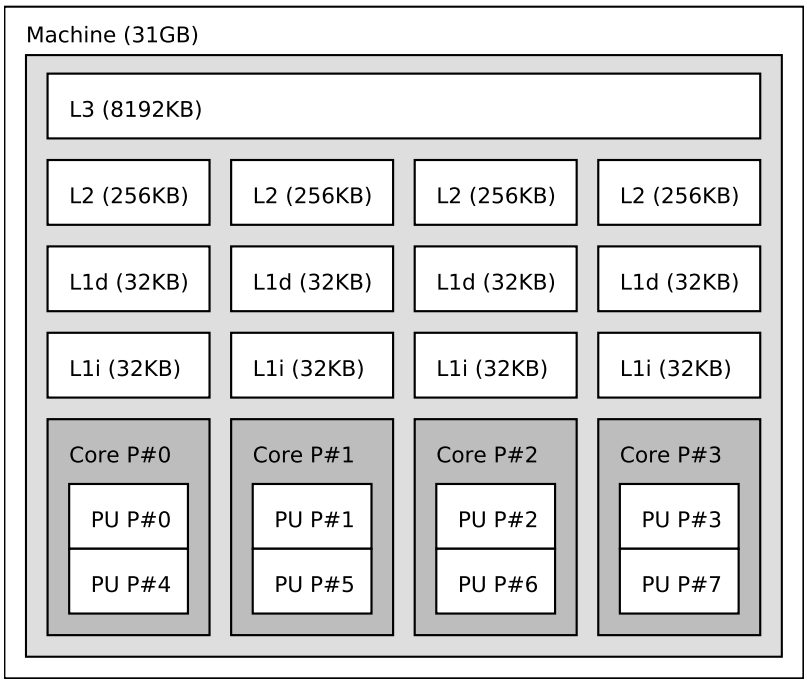


Figure 2.3.: Memory organization in a Intel i7-4770 Haswell processor

2.2.3. Hierarchical Organization of Memory

The gap in performance between processor and main memory started from early 80s to widen deeply in favor of processors. Since the processor speed is much faster than the memory access time, the processor would spend most of the time idle when requesting data from main memory (DRAM). In order to solve this problem, computer architects introduced cache memories between the CPU and the main memory. Caches are small memory pools built using static-RAM (SRAM) technology, thus have very low access times but they are also very expensive.

Since cache memory is expensive it is organized hierarchically. Lower level cache (closer to the CPU) is faster, but smaller and more expensive than the cache(s) situated in higher level(s). The size as well as the latency overhead of the hierarchical memory system in a modern processor is shown in Table 2.2.

Note that L1, L2 caches are typically located on the CPU die while L3 is placed on the system board as it is shared between the CPU cores in a multi-core processor. Further, L1 cache is typically divided for data (L1d) and instructions (L1i). An illustration depicting the hierarchical memory organization in a multi-core processor is shown in Figure 2.3.

Caches work adhering to the *temporal locality principle* [27], which means that caches hold the most recently accessed data. The programs are more likely to access again the recently accessed data, and due to the temporal locality principle, the data will be quickly found in the cache and this way resulting in a *cache hit*. If otherwise, a *cache miss* occurs and the data item has to be brought-in from the memory (DRAM). More specifically, the CPU will first look-up for a word in the L1 cache, and if not found, it continues searching in the L2 cache, and if not there, it looks it up in the L3 cache and finally in the main memory (DRAM). Each transition of the search from one level to another (deeper) level of memory induces significantly higher look-up costs, as shown in Table 2.2. Programs should therefore be designed with the memory hierarchy in mind in order to minimize the expensive memory traffic between the CPU and DRAM.

For efficiency reasons, the unit of transfer between the cache and the memory is a block or a *cache line* at the time. The block is typically 64 bytes long (a sequence of words) and in an event of a cache miss, a block of 64 bytes is transfered from memory (or a higher level cache) into the cache. There are three different placement schemes when it comes to placing a block into the cache thus leading to the notion of *cache associativity*:

- *n-way set associative* cache is divided into sets, where set is a group of n blocks of memory (or cache lines). A cache line is first mapped to a set, and then within the set it can be placed anywhere. In order to find a cache line in the cache, the set where the cache line could belong to is first computed, and within the set it is searched for the cache line in parallel. The set is found according to the address of the data [27]:

  $$\text{(cache line address) mod (number of sets in cache)},$$

- *direct mapping* cache contains sets able to hold only one cache line, therefore a cache line is always placed in the same location within a set,

- *fully associative* cache contains only one set, thus a cache line can be placed anywhere within the cache.

2.2.4. Virtual Memory

It is rather inefficient to allocate the entire memory space for each process as many processes use only a portion of their allocated address space. The problem of sharing the physical memory among processes is handled by the operating system by means of the *virtual memory*.

Virtual memory divides the physical memory into *pages* and allocates them to each running process. Such an allocation scheme provides protection; each

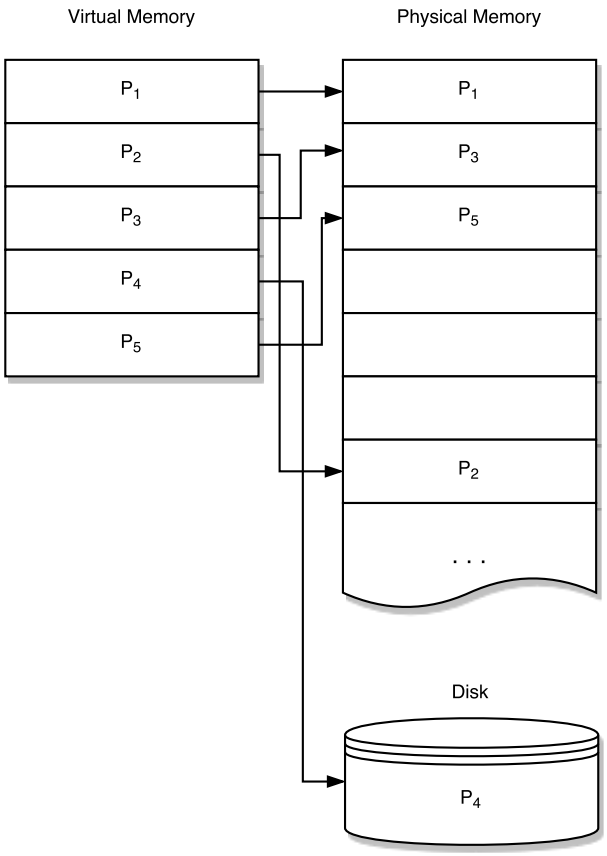


Figure 2.4.: Illustration of the contiguous virtual memory on the left and its mapping into the physical memory and the disk on the right

process can operate only over its allocated space and it cannot access the address space allocated to other processes. Further, by means of the virtual memory, the operating system can assign to processes more virtual memory than the available physical memory. In the sight of the process its assigned (virtual) memory is contiguous, but in reality its (virtual) memory space can be mapped to pages scattered across different locations in the main memory including the disk as well. Figure 2.4 illustrates the mapping of the virtual address space to the physical address. The page size depends on the processor architecture and is typically set to 4 KB, but larger pages are also supported.

Besides managing the physical memory and protection of the address space, virtual memory offers another benefit, it allows loading the same program on any physical memory location by means of *relocation* [27]. That is, the physical memory of a program can be placed anywhere in the main memory or disk and only the mapping of the virtual address space to physical memory need to be updated.

Virtual memory as seen by the process needs to be translated into the physical memory which is used by the hardware. This job is handled by the operating system by utilizing a data structure known as *page table*. That is, page table allows for translating virtual addresses to physical addresses. Since accessing

the page table for each translation request is an expensive operation, there exists special cache memory dedicated for holding entries of this structure, called *translation lookaside buffer* (TLB). Details on this structure are given in the following subsection.

2.2.5. Translation Lookaside Buffer

Page tables are typically very large and therefore stored in main memory. A request to translate a virtual memory address to physical memory address takes two memory accesses. The first one is to query the page table (a process called *page walk*) and the second one to access the actual data. In order to spare the traffic to main memory, most recent address translations are kept in a special cache called translation lookaside buffer (TLB). Just like the regular cache, the TLB is also hierarchical (L1, L2, L3), and divided for data (D-TLB) and instructions (I-TLB).

If the entry for an address translation request cannot be found in the TLB, an event known as TLB *miss* occurs. The TLB miss event triggers a page table lookup which is an expensive operation as it amounts to reading a number of memory locations in the page table in order to determine the physical address required by the process. Once the physical address is determined, it is then stored in the TLB, such that future memory translation requests result with a "TLB hit" and this way the expensive page lookups are avoided.

TLB is a very scarce resource, e.g., in Intel Haswell i7-4770 processor, L1 TLB has a capacity of only 64 entries and is 4-way set associative. Further, L1 TLB is split into the TLB for program addresses (I-TLB) and for data addresses (D-TLB). That is, the D-TLB and I-TLB have a capacity of only 32 entries each. Such a small L1 TLB capacity means that the new incoming translation requests if not present in the TLB evict older entries—as in TLB are kept only the most recent entries, according to the temporal locality principle—and this way causing expensive page lookups. Further, equally aligned addresses may also cause expensive page lookups as they *mutually evict* entries of one another in the TLB, even if the TLB capacity is not exhausted. This is due to the limited associativity typically found in TLBs. We show in Chapter 3 how the mutual eviction of equally aligned addresses affect a column store and how this problem can be alleviated.

2.3. Storage Layouts for a Main Memory Database System

Conceptually, database tables are two dimensional structures; columns representing the attribute values whereas rows represent the data about each entity individually. The conceptual design, however, differs from the physical design: the two dimensional tables need to be mapped to a one dimensional data structures, which are then stored in the storage medium (e.g., disks, RAM).

In the database world, there exists two major storage layouts: 1) the row-layout or the n-ary storage model (NSM) which stores the tables in a row-

by-row fashion, and 2) the columnar-layout or the decomposed storage model (DSM) [15], which stores the tables in a column-by-column fashion. Both storage layouts are prevalent in commercial databases. In the following subsections, we give more details on each respective storage layout.

2.3.1. Row Stores - NSM Layout

In NSM (N-ary Storage Model) storage layout, relations are stored in a row-by-row fashion, where each row corresponds to a tuple. That is, all attribute values of a tuple are stored closely together.

An example C++ code fragment of our Movie database in the row storage layout is shown below:

```
struct movie_t {
  std::string _movie;
  int         _year;
  double      _rating;
};
std::vector<movie_t> Movies;
```

Row stores were designed with the goal of handling OLTP workloads. In such workloads the records are read/updated in an entity granularity, e.g., update a customer's bank balance, transfer funds from one customer to another, etc. Since the data in a row store are stored in tuple-wise fashion, row stores have a low tuple reconstruction costs due to the co-location of attribute values. On the other hand, sequentially scanning a single attribute (or few attributes) in a row store is an expensive operation as the entire rows have to be fetched from the main memory/disks, thus resulting with a suboptimal utilization of the memory bandwidth. In addition, caches are loaded with unnecessary attribute values. Section 2.3.2 show experimentally that read operations over a single attribute in a row store are much more expensive than in a column store.

2.3.2. Column Stores - DSM Layout

In contrast to row stores, column stores partition relations vertically into binary relations, where each such binary relation corresponds to an individual attribute. That is, a relation with n attributes is decomposed into n binary relations. Binary relations in turn are composed of two attributes: the *surrogate*, and the *attribute*. Note that surrogates (i.e., rids) can be left virtual, they do not have to be explicitly materialized. This storage scheme is known as the DSM (Decomposition Storage Model) [15] or vertically partitioned storage layout. MonetDB [5] is a notable system adopting this storage layout.

Figure 2.5 depicts our example of Movies relation and its decomposition into the DSM storage layout. The original relation can be reconstructed by means of joins on rids.

An example C++ code snippet for our example Movie database in the columnar storage layout is shown below.

```
struct Movie {
  std::vector<std::string> _movie;
```

Movie	Year	IMDB rating
The Godfather	1972	9.2
The Dark Knight	2008	8.9
A Beautiful Mind	2001	8.2

rid	Movie
1	The Godfather
2	The Dark Knight
3	A Beautiful Mind

rid	Year
1	1972
2	2008
3	2001

rid	IMDB rating
1	9.2
2	8.9
3	8.2

Figure 2.5.: Vertical partitioning of the Movies relation

```
    std::vector<int>          _year;
    std::vector<double>       _rating;
};
```

Column stores open a possibility for a fine-grained (selective) representation; a column can be stored in multiple sort orders, thus allowing for better compression schemes. In general, column stores yield very good compression ratios (e.g., see [2]) as the data of each attribute are kept close together, and further, they are of the same type, thus reducing the entropy.

Column stores are especially attractive for applications in the Business Intelligence (BI) domain. In contrast to row stores where queries operate on an entity granularity, queries in the BI domain are typically long running queries (known as OLAP queries) producing data summaries over a large set of records but touch only few attributes, e.g., find the average balance of all customers for 2016. Column stores offer an attractive query execution environment for OLAP queries, as they allow fetching from the memory/disk only the columns used in the query (and not entire rows), and this translates to reading less data, thus the better utilization of the memory bandwidth. Column data items are much smaller in width (compared to reading entire rows as it is the case with row stores), therefore they fit nicely into CPU caches and this way allow for a reduced cache miss ratio.

The scan operator is a fundamental operator in a database system as all other operators are built on top of the scan operator. Scan operations over columns in column stores are extremely efficient operations due to the locality of data items in the respective columns. A column scan operation exhibits a sequential access pattern, enabling the CPU prefetcher to bring into CPU cache the column items in advance, and this way minimizing the memory access latency. To show this, we have performed a small experiment showing the time it takes a sequential scan over a single attribute in a row store vs. column store.

For this experiment, we have used our movie database; for the row store, the

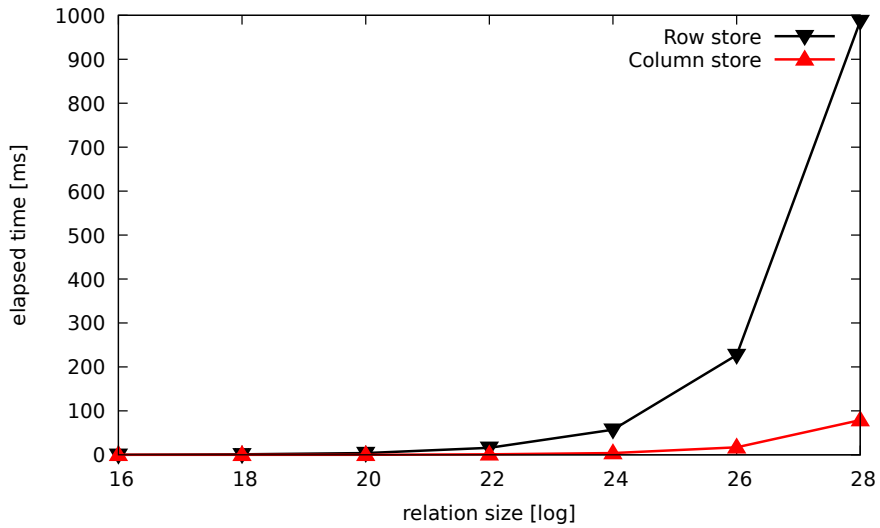


Figure 2.6.: Scan time in a row store and a column store over a single attribute

data were materialized in main memory in an instance of the movie structure shown in the code snippet in Section 2.3.1 and for the column store, in an instance of the structure shown in the code snippet above.

A number of relations were generated with cardinalities starting from 2^{16} and up to 2^{28}. The attribute values for the relation `Movies` were picked randomly from a pool of movie data collected from IMDb. The query used in this experiment projects the values of the attribute `year`: $\pi_{year}(Movies)$. The query was hand-coded in C++, and executed over both, the row and column store. The runtime of the scan operations for both, the row store and the column store were measured over all the relation sizes $2^{16} - 2^{28}$. The experiment was run in a machine with Intel Xeon E5-2690 v2 3.00GHz processor and 256 GB of main memory. The results of this experiment are shown in Figure 2.6.

As it can be seen in Figure 2.6, the scan operation in the column store is by far more efficient than the same operation over the row store. For example, if we look at the relation cardinality of 2^{28}, the scan operation in the column store is a factor of 12 cheaper than the same operation in the row store. Such a large difference in the runtime comes mainly from the fact that only the values of the attribute `year` were required in the query. In the column store, the scan operator iterates over the items of a single vector; the column items of the attribute `year` are narrow in width (i.e., 32 bits), therefore they fit nicely into the CPU cache lines. In contrast to the column store, in the row store the tuples are much wider—they contain the values of all the attributes in the scanned relation—thus causing expensive cache misses. In column stores, the cache lines are filled with consecutive data items from the particular column being scanned, allowing for an optimal utilization of the cache. That is, the cache lines are not polluted with irrelevant data belonging to other attributes which are not required in a query. In row stores, all the attribute values of a tuple have to be brought into the cache lines even if the values of only a single

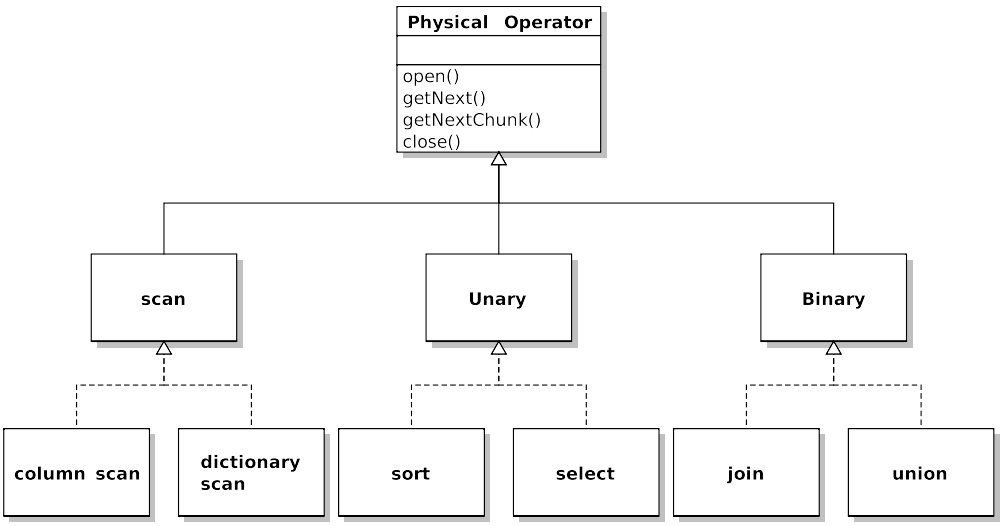


Figure 2.7.: An UML graph of physical operators in SystemTx

attribute are required, therefore, the cache lines are polluted with unnecessary data items. Of course, our experiment was biased towards column stores as the query projected the values from a single attribute only. Nevertheless, our experiment shows the superiority of column stores for applications that require reading large amount of data but from rather few columns, as it is the case with OLAP workloads.

2.4. Iterator Model

Query execution in traditional database systems consist of physical operators that implement an interface consisting of three primitive virtual functions: open(), getNext(), and close(). This is known as the *volcano-style* [23] iterator-based query execution, or simply the pull model.

An UML class diagram of operators that implement these virtual functions in SystemTx has been depicted in Fig. 2.7. We separate the operators in three groups: scan operators, which basically scan the source of tuples, the second group consists of operators that consume tuples from a single input. The third and the final group consists of operators that consume tuples from two inputs, e.g., the left and the right input in a join operator.

A volcano-style query execution works by first having the root operator call the function open() on all its children operators all the way to the leaf operators. In response to this call, each operator in the operator tree initializes its resources. A consumer operator, in this case the root operator pulls the tuples from its children operators by means of the function call getNext(). After the entire stream of tuples has been processed, the root operator propagates the function call close() to all its children operators, and as effect, all the operators receiving this call close their resources and release their buffers. This iterator model (pull model) is illustrated in Figure 2.8.

The dual of the pull model is the push mode. The push data flow model differs

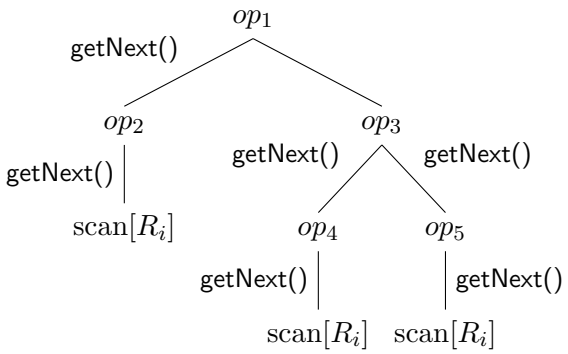


Figure 2.8.: Illustration of the pull-based iterator model

from the pull data flow model, in that the *stream of tuples* are not pulled but rather pushed towards consumer operators by their children operators all the way to the root (top-most) operator. It has been shown in [52] that the push model allows for a better code and exhibits better data locality. We have, therefore, implemented the push model in our system.

In the literature, tuple producing operators are categorized into three groups: operators that produce *tuple-at-a-time*, operators that produce a block of tuples – *chunk-at-a-time*, and the operators that materialize entire columns - *column-at-a-time*. In the following subsections, we briefly explain these groups of operators.

Tuple-at-a-time

Tuple-at-a-time is an iterator model used commonly in traditional database systems, whereby operators produce a single tuple for each getNext() function call. In this approach, operators do not materialize tuples, but they route them to their parent operator (consumer operators). This is known in database terminology as *pipelining*.

On the other hand, *pipeline breakers* are those operators that materialize their tuples before passing them to the next operator. A good example of a pipeline breaker is the hash-join operator. The tuples from one of its sides (recall that join operators are binary operators) are materialized in a hashmap structure, therefore the pipeline is broken. The tuples from the other side are then probed against the hashmap, and only if they qualify after this step, they flow to the next operator.

The main drawback with the tuple-at-the-time execution paradigm is that it incurs high interpretation overhead. Depending on the operator tree size and the selectivity of predicates, an arbitrary large number of function calls take place before a tuple is produced and has reached the root operator. That is, the function getNext() can be called million times or more to process a column, depending on the column size. In addition, each function invocation (e.g., getNext()) corresponds to a look-up in the *virtual function table*, thus adding more costs. As the function call getNext() is routed from one operator to another,

the CPU cache has to be flushed and reloaded with operator specific instructions as well as operator data. Inadvertently, this leads to high cache miss-ratio, and causes expensive memory stalls.

Ailamaki et. al [3] show that 90 % of memory stalls in database systems are caused by first level (L1) I-Cache and second level (L2) D-Cache misses. First level I-Cache size is relatively small (in range of 4 - 32 KB), hence instruction misses occur often in the tuple-at-a-time iterator model even for a small operator tree size.

Chunk-at-a-time

Chunk-at-a-time is an execution scheme where instead of pipelining a single tuple, operators fill a chunk with tuples and pass an iterator (which is merely a memory reference) to their parent operator. An iterator in this context is a pointer that points to the chunk's start address. We refer to the chunk as buffer, which has a *start* position, a *size* attribute, and an *end* position.

Chunk-at-a-time scheme has the advantage that only a memory address is routed from one operator to another instead of expensive copies of chunks. However, operators in this approach have to break the pipeline, due to buffer materialization, thus costing additional memory. The advantage of the chunk-at-a-time approach however is that it allows for *block-oriented processing* of tuples, thus reducing significantly the number of getNext() function calls. The latter is replaced by the getNextBlock() function call which significantly amortizes the costs of the function call getNext(), as getNextBlock() is called on chunk-basis and not on tuple-basis, as it is the case with getNext(). In chunk-at-a-time, the consumer operators iterate over tuples in a chunk in a tight loop, e.g.:

```
for(Iterator* it = chunk.begin(); it != chunk.end(); ++it) {
  // do smth with a tuple, e.g., print it
  print(*it);
}
```

The processing of tuples in a tight-loop opens doors to the efficient *vectorized execution*, as such tuple iteration is not interrupted by the function calls getNext(), i.e., more valuable CPU time is spent on operating over values than on function call overhead [7, 64]. This scheme opens doors for other optimizations such as loop-pipelining, automatic SIMD code generation by the compiler, less data cache misses due to high data locality (cache lines are filled with data from one chunk, thus less memory traffic). A notable system implementing this execution scheme is VectorWise [66].

Chunks hold relatively small number of (cache resident) items and are materialized incrementally. Chunks are implemented as an array (vector) in the actual code. The array size is a system parameter and it is set in accordance to the cache size, such that the arrays can fit into the CPUs L2 D-Cache. If arrays do not fit into the CPUs cache, expensive memory traffic between cache and main memory is caused. Such traffic forms a major bottleneck in main memory database systems. This is well illustrated in the Figure 2.9. According to the experiments in [65], as the vector size increases beyond 2K elements, they start

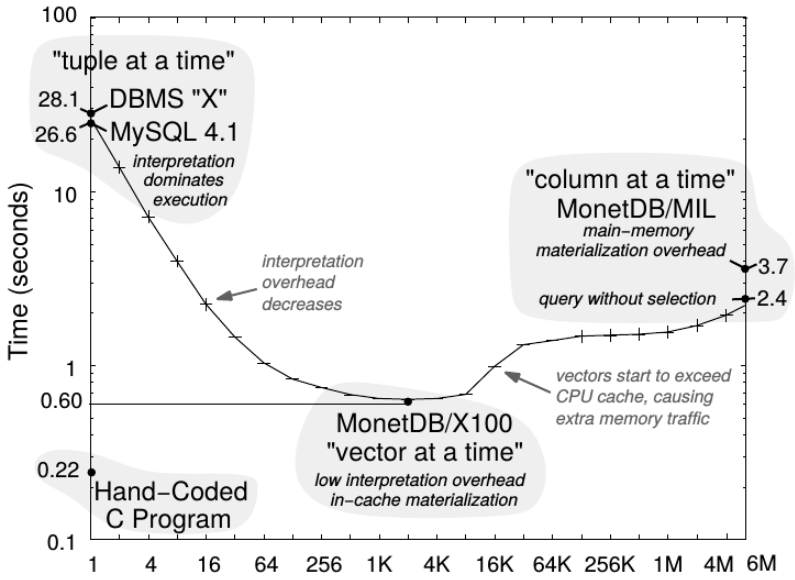


Figure 2.9.: Illustration of tuple processing schemes. The x-axis denotes the vector size. Figure taken from [65].

spilling into main memory (they don't fit any longer into the CPU cache), thus expensive memory traffic is caused.

Column-at-a-time

Systems like MonetDB [6] have taken the other extreme and materialize intermediate results - entire columns.

Materializing entire columns leads to the advantage that only one function call is required for processing all the tuples by a single operator. Just as in chunk-at-a-time execution model, this execution scheme opens ways for optimizations such as loop-unrolling, automatic SIMD instruction generation by compilers, etc. That is, tuple interpretation overhead is significantly reduced, however, at the price of high memory consumption. When working with large data sets, expensive memory traffic is caused by each operator, as operators will spend significant time writing into memory, and the intermediate results won't fit into the CPU cache, see Fig. 2.9. This problem is further exacerbated with multi-CPUs which share their memory, as shown in [64].

3. SystemTx

In this chapter, we first present our main memory column oriented database system coined SystemTx. After a brief introduction of SystemTx, a section follows which presents the initial storage layout used in SystemTx. The initial storage layout was then replaced by a new storage layout as it caused high TLB miss rates when scanning multiple columns sequentially. The underpinnings of the new storage layout are the topic of the last section of this chapter.

3.1. Introduction

Although SystemTx is a main memory column store, we use rows/tuples as a representation of intermediate results. This allows for better cache locality during the evaluation of expressions. Second, we implemented the *push-based* model, as it allows for better code and exhibits better data locality [52]. In a push-based model, each algebraic operator implements an interface with `init`, `step`, and `close` functions. The `step` function is the most important. It accepts an input tuple, processes it, and passes it to the consumer operator up the tree via calling the step function of the consumer.

```
TX_Scan::run() {
  for(i=0; i<|R|; ++i) {
    t.rid=i; t.ap++; t.bp++; t.cp++ ...
    consumer.step(t);
  }
}
```

The RID variable and the column pointers in tuple t are maintained by the scan operator (as depicted in the pseudo code above). This way, they point to the correct column values, and upon request, such column values can be fetched by means of the map operator, as shown in the code snippets below.

In SystemTx, there exist two ways of dereferencing (accessing) column values. The first method accesses column values based on row identifiers (RIDs). In pseudocode, this reads as:

```
Tx_MAP1::step(t) {
  t.A = R.A[t.rid];
  t.B = R.B[t.rid];
  t.C = R.C[t.rid];
  ...
  consumer.step(t);
}
```

The second method accesses column values based on column pointers

```
Tx_MAP2::step(t) {
  t.A = *(t.ap);
  t.B = *(t.bp);
  t.C = *(t.cp);
  ...
  consumer.step(t);
}
```

The column values are also stored in the tuple t, which is then passed to the next operator (consumer) in the operator tree.

The selection operator simply pipelines the qualifying tuples to its consumer operator.

```
Tx_Select::step(t) {
  if(p(t)) consumer.step(t);
}
```

3.1.1. Physical operators in SystemTx

In this section, we present physical operators implemented in the SystemTx that are of relevance to this thesis work.

Sequential scan operator `scan`(R) scans an input relation R by means of a tuple t. The tuple t contains an attribute named RID, which represents the row identifier and pointers to columns of R; these pointers are offsets to the respective column values. The number of pointers in tuple t is query dependent, that is, for each attribute required in a query, there is a pointer to the values of that attribute (i.e., column).

The scan operator iterates over all "tuples" by incrementing the pointers in t and the RID variable. The tuple t is pushed iteratively to the consumer operator via the consumer's `step` method call as shown in the pseudo of the previous section.

Before we present the bypass selection operator, let us briefly recall the regular selection operator defined in Section 2.1. The selection operator $\sigma_p(e) := \{\, t \,|\, t \in e \wedge p(e)\}$ filters out all the tuples that do not satisfy the predicate p. The tuples that pass the predicate are passed up higher in the tree to the next operator.

Bypass selection operator $\sigma_p^+(e) := \{\, t \,|\, t \in e \wedge p(e)\}$ **and** $\sigma_p^-(e) := e - \sigma_p^+(e) \equiv \{\, t \,|\, t \in e \wedge \neg p(e)\}$ in contrast to the regular selection operator bifurcates the input stream into two disjoint streams; the true stream denoted by $\sigma_p^+(e)$, and the false stream denoted by $\sigma_p^-(e)$, respectively. To this end, the two output streams are finally merged by the union operator $\uplus$ (without an expensive duplicate elimination, see Section 6.4), sitting on top of the plan.

One should not think of the bypass selection operator as two operators, where one produces the true stream and the other one the false stream of tuples. This operator is implemented as a single operator $\sigma^{\pm}$ and produces both streams simultaneously. The benefits of the bypass selection operator are shown in

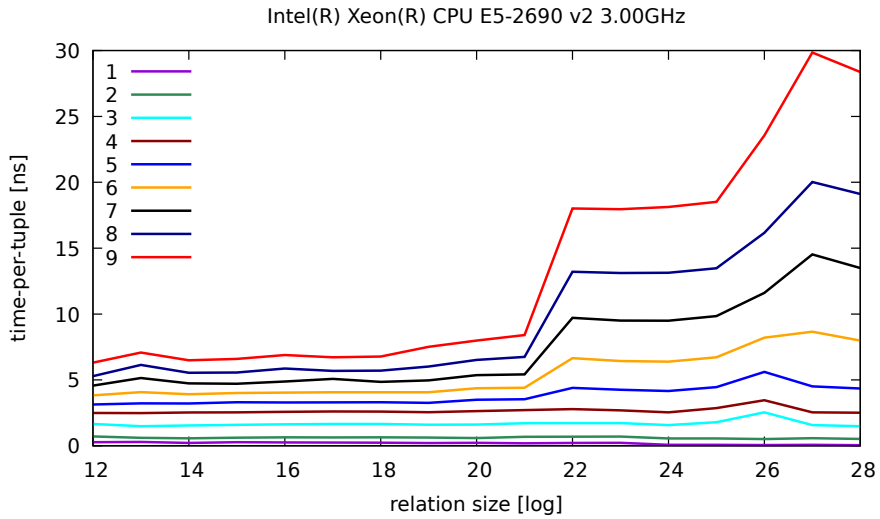


Figure 3.1.: Column access costs over the initial storage layout

Chapter 6. In the following, the pseudocode for bypass selection operator is shown.

```
Tx_BYPSelect::step(t) {
  if(p(t)) {
    consumer_true.step(t);
  } else {
    consumer_false.step(t);
  }
}
```

3.2. A Bad Storage Layout

In this section we present our initial storage layout used in our in-memory database system – SystemTx. In this storage layout relations are entirely kept in main memory, and they are vertically partitioned (DSM scheme), that is, attribute values of each attribute are stored in a separate column. Columns in turn are stored in separate vectors (i.e., arrays), just as in the example storage layout given in Section 2.3.2. A relation R may contain a number of such vectors depending on the number of attributes, whereby each vector represents an individual attribute of R.

One important cost factor in a main memory column store is the cost of the dereferenciation operator used for column access [1]. The scan time of course is proportional to the cardinality of the column. Thus, several relation sizes must be tested. We were interested in the costs of scanning multiple (i.e., $1, 2, \ldots, 9$) columns simultaneously. The measurements for such scans over our initial storage layout are contained in Figure 3.1. The x-axis is labeled by the base-2 logarithm of the relation's cardinality. The y-axis presents the access

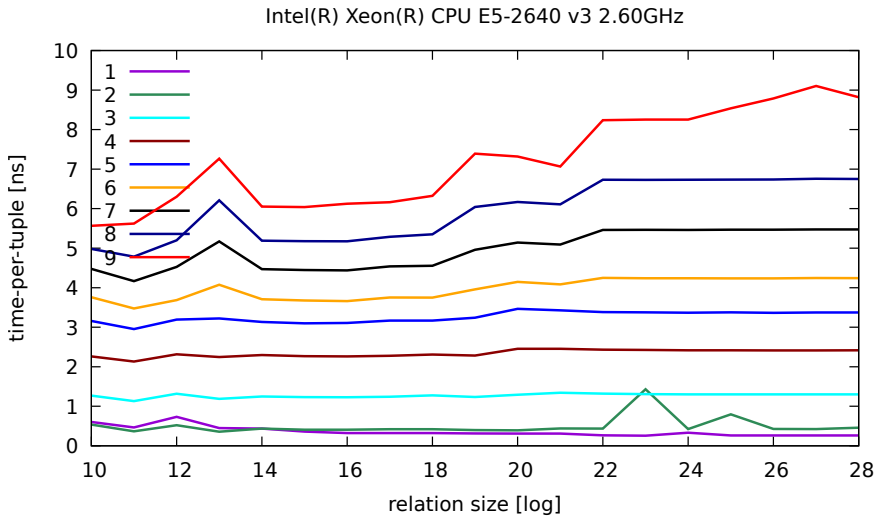


Figure 3.2.: The effect of DTLB misses on a column store

time per column and row in nanoseconds. Each of the curves 1-9 corresponds to 1-9 simultaneous column scans.

From the cost function point of view, these curves are sub-optimal. They give the impression that they cannot be easily approximated with a high precision, i.e., an approximation yielding a small q-error (cf. Chapter 4). And indeed, the visual impression turned out to be true. Further, note the steep increase from relation size 2^{21} to 2^{22}. To accurately model this steep ascent, additional relation sizes between these two points ($2^{21} - 2^{22}$) would have to be generated and scan costs on these would have to be measured. Clearly, this would have a profound negative impact on the calibration time of the cost model. Last but not least, we were not satisfied with the high latency in column accesses, which as shown in Figure 3.1 are in order of 30 ns for 9 columns. To this end, our initial storage layout proved to be a bad basis for a query execution engine.

An analysis of the cause of this bad behavior revealed the following. The reason for the steep increase in the latency times is the high rate of L1 DTLB misses (details are given in Sec. 2.2.5). To understand why this effect only shows for more than 4 simultaneously scanned columns, it is important to know that the processor used is an Intel Xeon E5-2690 v2 3.00GHz processor. This processor has four prefetchers, explaining that there is virtually no difference in time between scanning 1 column and scanning 4 columns simultaneously. The second important piece of information is that the processor has a 4-way associative L1 DTLB. This explains why the curves go higher and higher for more than 4 simultaneous column scans. That is, the fixed stride in such column accesses operations lead to the eviction of DTLB entries. The last point is that the steep increase occurs only for relations with cardinalities larger than 2^{21}. This is explained by the small L1 DTLB size.

We have repeated the same experiment using newer hardware (Haswell and Skylake XEON processors). The results of this experiment are shown in Fig-

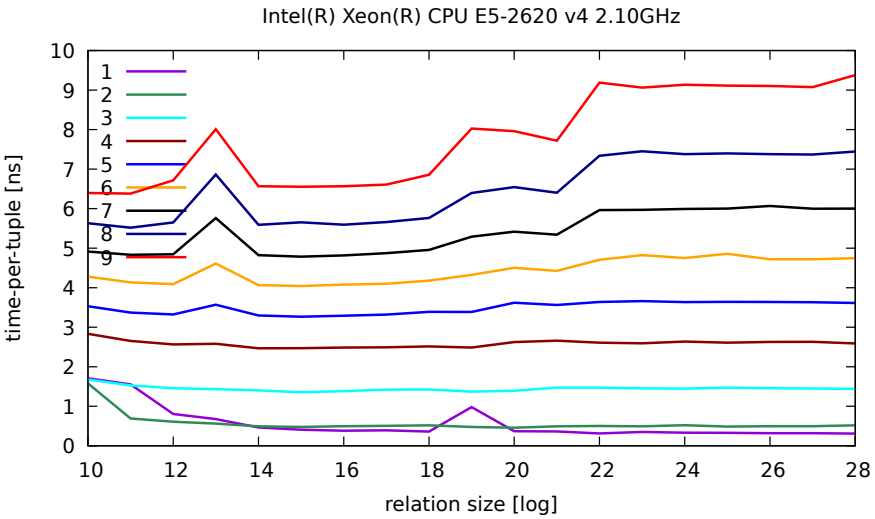


Figure 3.3.: The effect of DTLB misses on a column store

ures 3.2-3.3. In the new experiment with the new hardware, we see that the effects of the DTLB misses are not as severe as in the previous experiment. Nevertheless, in the next subsection we present a new storage layout which alleviates the negative effect of DTLB misses regardless of the hardware (new or old), and in addition, the scan operations become cheaper.

3.3. Storage Layout in SystemTx

Having found the reason for the steep ascend in access time, we were looking for the cause within the initial storage layout. In the initial storage layout, during bulk load and during restart, system allocates huge chunks of memory, holding, whenever possible, a whole column of a given relation. This results in scan strides which suffer badly from L1 DTLB misses. Thus, we decided to implement a new storage layout. In the new storage layout, we changed (among other things we did not like either) the memory allocation strategy. Instead of allocating huge chunks for each column, we allocate multiple smaller chunks of memory for each column. These chunks are not of fixed size but instead are able to contain a fixed number of attribute values. Further, the allocation strategy goes round robin on the columns. This is illustrated in Figure 3.4. The green rounded rectangles represent the logical columns, and the white rectangles represent the column chunks, which in the figure are accidentally all of the same size. The line with the arrows demonstrates the timeline of the chunk allocation process. Each column maintains offset pointers (in an array) to the beginning of each of its constituent chunks. Using these pointers we can iterate over items belonging to a column as if they were stored contiguously in the memory. Since the chunk's cardinality is known (is a global parameter), we do not risk overflowing the chunks when scanning columns. Upon scanning all the items belonging to a single chunk, we jump to the next chunk (belonging to

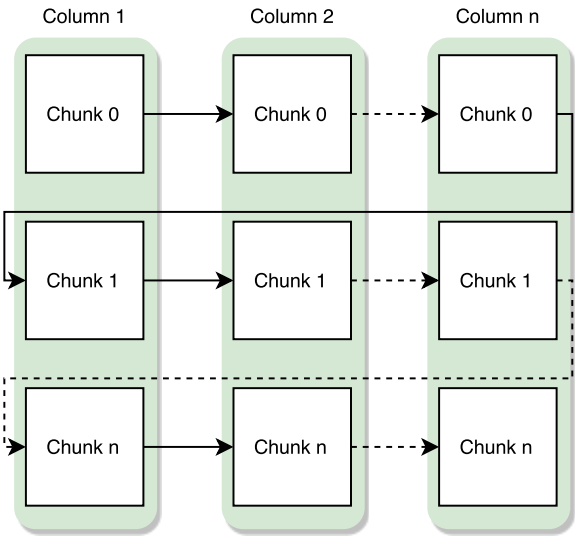


Figure 3.4.: Illustration of the chunk-wise storage layout in SystemTx

the column that we are currently iterating) by following the next chunk pointer. Technical details on the allocator are presented in Appendix A.1.

The benefits of the new storage layout for the old and new hardware can be seen in Figures 3.5–3.7. The curves are now more streamlined, therefore allowing for better approximation by cost functions (cf. Chapter 4), and further, the absolute column scan costs have also dropped as a side-effect, which we warmly welcomed.

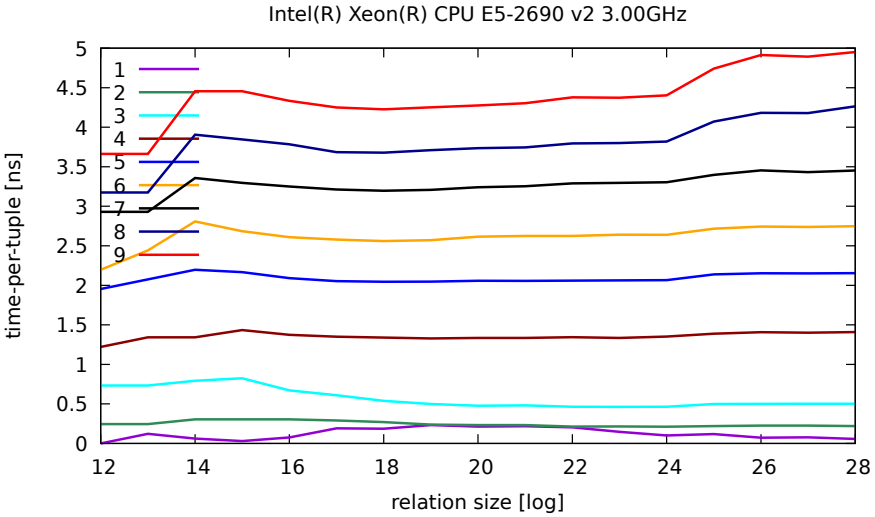


Figure 3.5.: Reduced DTLB miss effect using the new storage layout

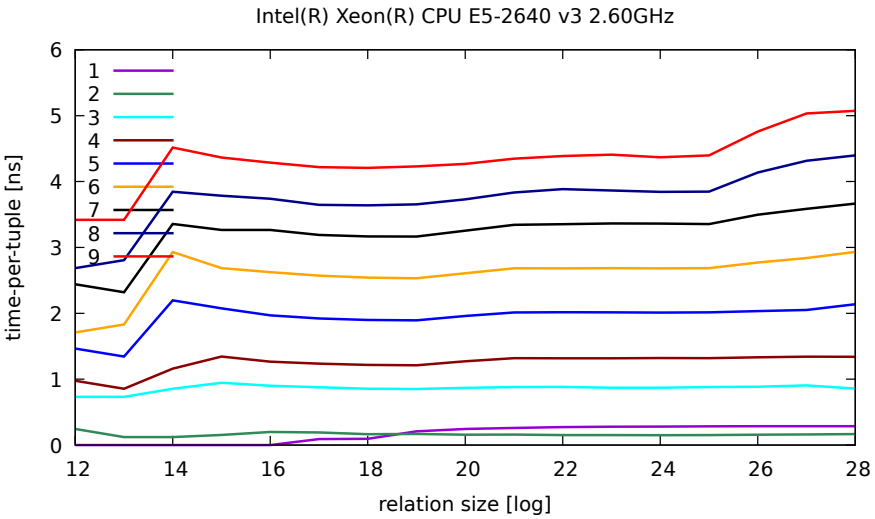


Figure 3.6.: Reduced DTLB miss effect using the new storage layout

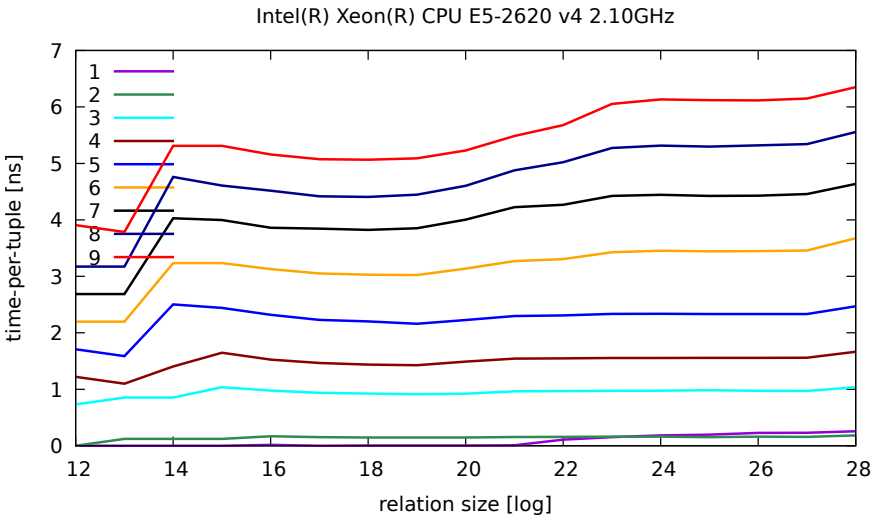


Figure 3.7.: Reduced DTLB miss effect using the new storage layout

4. Cost Estimation and Approximation

In this chapter, we present the cost model derived from our column store – SystemTx. Cost model presented in this chapter is used by the optimization algorithms presented later in Chapter 5 to Chapter 7.

The contents of this chapter were published in [33].

4.1. Introduction

The goal of the query optimizer is to find the optimal (i.e., the cheapest) plan from the space of all possible plans. Query optimizers discriminate the enumerated plans based on their costs. Since the cost metric plays a decisive role in finding the "cheapest" plan, it is important that our cost model closely resembles the true costs of a plan, i.e., the cost of an actual execution of the plan. The cheapest plan doesn't necessarily have to be the plan with lowest running time, it could be the one that minimizes the energy consumption (e.g., for databases running on handheld devices), or the time until the first tuple is produced. Nevertheless, in this chapter we assume that the cheapest plan is the one with the lowest total execution time.

In the new era of emerging main memory databases, the role of I/O costs has diminished, whereas the CPU costs have taken the dominating role. In main memory databases, CPU architectural characteristics such as the branch misprediction penalty can outweigh by far the costs of simple comparisons, therefore they should be well estimated by a cost model. Further, costs such as accessing column values, incrementing iterators, tuple pipelining etc., play an equally important role in a main memory database system.

In this chapter, we tackle two subproblems related to the cost model. First, we establish the cost functions, and second, we show how to obtain the parameters for the cost functions.

Cost functions cannot model 100% error-free the true costs of a plan due to the speculative nature of modern CPUs, hierarchical memory etc. However, we should strive to minimize the error if the goal is to find the best plan, i.e., the plan with the lowest execution costs. The majority of the cost function in the literature minimize the ℓ_2 error. Since ℓ_2 does not provide bounds on error, it is not suited for query optimization. In this thesis, we approximate the cost functions under the q-error. Approximation under the q-error, it turn, provides us bounds on the quality of the approximation. In the light of q-error, we present two important findings: (1) we show that if our cost function is precise up to a factor q, then the plan picked by our optimizer under this (erroneous) cost function is at *most* a factor of q^2 far from the optimal plan. That is, we show a direct link between the error of the cost functions and the plan quality.

Although a direct link between the error of the cardinality estimation and the plan quality has been shown in [48], we show for the first time that there is also a direct link between the error of the cost function and the plan quality. (2) We show that if the q-error is bounded by a small q, then the best plan picked by the optimizer is still the optimal plan.

Cost functions depend on cardinalities. In Chapter 8, we present an efficient method based on sampling which can be used to find the selectivities for all the subsets of predicates given in a query.

The rest of the chapter is organized as follows. Related work is described in Section 4.2. In Section 4.3 we define the q-error, whereas in Section 4.4 we show its theoretical implications for our cost model. In Section 4.5 we state the cost functions for the physical operators scan, selection (σ), and the map (χ) operator. Additionally, we provide cost functions for the evaluation of conjunctions $p_1 \wedge p_2$ of predicates by expressions either of the form $p_1 \& p_2$ or of the form $p_1 \&\& p_2$. The benefits of each of these alternatives were discussed in depth in Section 2.2.2. Section 4.5 also includes the cost functions for memory accesses and branch misprediction. In Section 4.6 we present our approximation framework which is used to obtain the parameters for our cost functions, whereas the validation of our cost model is shown in Section 4.7.

4.2. Related Work

An accurate generic cost model for main memory database is presented by Manegold et al. [40]. Their cost model covers the hierarchical memory model in modern processors (cf. Section 2.2); it accounts for architectural characteristics such as cache and TLB misses. It further distinguishes between sequential and random access patterns as obviously they exhibit different costs. The authors in [40] provide a calibrator[1] tool, which can be used to extract parameters for their cost model. Such parameters include the size of the CPU cache(s), cache line size, memory access time, TLB miss latencies etc. Despite being quite precise, their cost model does not account for the branch misprediction penalty, which in turn, is an important cost factor for main memory databases as we will show in this chapter.

The work in [54] extends the cost model in [40]. It adapts it for the partially decomposed storage model (PDSM) for main memory databases [24] and JIT compiled queries. JIT compiled queries contain many nested interleaved physical operators which result with complex memory access patterns, thus making the cost model non-trivial to devise. In this work, the cost of branch misprediction penalty for queries over the generated partitions are not accounted for. However, the branch misprediction penalty in Intel processors outweighs both L1 and L2 data cache misses in terms of CPU cycles[2].

Another cost model for partially decomposed storage model for main memory databases was presented by Grund et al. in [24]. They first present a cost model which aims to accurately model the cache miss rate for a given

[1] http://homepages.cwi.nl/~manegold/Calibrator/
[2] http://www.7-cpu.com/cpu/Haswell.html

partitioning scheme. Having a cost model that models the cache miss rate, they present a database design algorithm for finding a partitioning scheme for a given workload, which minimizes the cache miss rate. The cost of the branch misprediction penalty for queries over the generated partitions is not accounted for.

To the best of our knowledge, the work by Ross [56] is the first one that provides a cost model which includes the branch misprediction component. However, the cost model in [56] is rather simple, it assumes a perfect branch predictor in a CPU. That is, it assumes that a branch will be taken whenever the selectivity of an atomic predicate $sel(p) > 0.5$, or otherwise, i.e., when $sel(p) \leq 0.5$. In reality, the branch predictor in CPU is a much more complex piece of hardware and using such a simple prediction model, one cannot truly model it. We take a different approach, we approximate the branch misprediction costs over the entire selectivity range. Our approximation method relies on the recent advances in approximation theory, thereby yielding a cost functions which models branch misprediction very accurately, as confirmed by the validation of our cost functions in Section 4.7.

4.3. The Convex Paranorm: Q-paranorm

In this section, we give some background information on the Q-paranorm, in order to understand the error metric used in this work. A detailed exposition of this subject is given in [45, 46].

The norms $\ell_1, \ell_2, \ell_\infty$ are generally well known and well covered in the literature, however that is not the case with ℓ_q. For $x \in \mathbb{R}$, the Q-paranorm $(\|.\|_Q)$ is defined as follows [45]:

$$\|.\|_Q = \begin{cases} \infty & \text{if } x \leq 0 \\ 1/x & \text{if } 0 < x \leq 1 \\ x & \text{if } 1 \leq x \end{cases}$$

The multivariate case is defined by simply taking the maximum over all elements of x:

$$\|x\|_Q = \max_{i=1}^{n} \|x_i\|_Q$$

The Q-paranorm is denoted by ℓ_q. In similar fashion as with the norm, we now give the definition of the paranorm.

Definition 4.3.1. (paranorm) Let V be a linear space, and let $\|.\| : V \to \mathbb{R}$, where $\|.\|$ is a real-valued function over V, such that

1. $\|w\| \geq 0$
2. $\|w + v\| \leq \|w\| + \|v\|$

Then $\|.\|$ defines a *paranorm* in V.

name	definition		minimizes error
median	$\begin{cases} x_{(n+1)/2} & \text{n is odd} \\ (x_{x/2} + x_{n/2+1})/2 & \text{n is even} \end{cases}$		$\mathcal{E}_1 = \sum_i \lvert x_i - \hat{x} \rvert$
mean	$\frac{1}{n} \sum_i x_i$		$\mathcal{E}_2 = \sqrt{\sum_i (x_i - \hat{x})^2}$
middle	$(\max(X) + \min(X))/2$		$\mathcal{E}_\infty = \max_i \lvert x_i - \hat{x} \rvert$
q-middle	$\sqrt{\max(X) \min(X)}$		$\mathcal{E}_q = \max_i \max\{x_i/\hat{x}, \hat{x}/x_i\}$

Table 4.1.: Equations for approximating a set of numbers and the error they minimize

The distance of two vectors $w, v \in \mathbb{R}^n$ under ℓ_q is defined as follows

$$d_q(w, v) = \|w/v\|_Q, v > 0$$

and the w/v for the vectors w and v is defined simply as element-wise division:

$$w/v = (w_1/v_1, w_2/v_2, \ldots, w_n/v_n)^\intercal, v_i > 0$$

The symbol $\intercal$ denotes the transposition.

Given the background information on vector norms, we state formally our approximation problem: For a matrix $A \in \mathbb{R}^{m \times n}$, and set of points (x_i, y_i), where $(1 \leq i \leq m)$, and $\vec{y} = (y_1, y_2, \ldots, y_n)$, we need to find a vector $\vec{c} \in \mathbb{R}^n$, so the distance $d(A\vec{c}, \vec{y})$ is minimal. The vector $\vec{c}$ is then called the *best approximation*, or the *solution*.

Our goal is to find the *best approximation* under ℓ_q, as we want to minimize the multiplicative error $\mathcal{E}_q$.

Let $x > 0$ be a value and $\hat{x} > 0$ be an estimate for it. Then, the *q-error* of the estimate $\hat{x}$ is defined as

$$\text{q-error}(\hat{x}) := \|\hat{x}/x\|_Q \,, \tag{4.1}$$

Thus, the q-error measures the factor by which the estimate $\hat{x}$ deviates from the true value x.

A strong argument for choosing the q-error as the error metric is that the q-error gives error bounds, which is not the case with ℓ_1 nor with ℓ_2. Linear regression for instance, minimizes ℓ_2, and since it gives no error bounds is not useful in the context of query optimization. This leads to the question why is the error bound important? If we have an error bound, then for a given estimate $\hat{x}$ we can derive the interval which with certainty contains the true value x. Even more importantly, error bounds enable us to establish a direct link between the quality of the cost function and the plan quality. More details will follow in the next section.

Let us exemplify this with a concrete example. Assume we have a set of points $Z = \{2, 4, 9\}$, and then we derive the following approximations for Z, according to the equations given in Table 4.1, thus yielding:

The error which is minimized by each approximation is given in the rightmost column of the Table 4.1. Out of the error metrics shown in the rightmost column of Table 4.1, only $\mathcal{E}_\infty$ and $\mathcal{E}_q$ give error bounds.

median	mean	middle	q-middle
4	5	5.5	4.2

Now lets derive the error bounds for $\mathcal{E}_\infty$. If $max(x) - min(x)$ defines the spread δ of x, it follows that for every value $x_i \in X$, where X is the set of values we wish to approximate:

$$m - \delta/2 \leq x_i \leq m + \delta/2$$

and m is the middle of X. This way we have an additive and symmetric error bound for all elements in the set X.

For our example set Z, we have $m = 5.5$ whereas $\delta = 9 - 2 = 7$. Thus all the elements $x_i \in Z$ are bounded by

$$5.5 - 7/2 \leq x_i \leq 5.5 + 7/2 \implies 2 \leq x_i \leq 9.$$

If we inspect the elements of our example set Z, the inequality shown above indeed holds for all the elements of Z.

In similar fashion as for $\mathcal{E}_\infty$, we now show how to derive the error bounds for the q-error too. The geometric spread [46] is defined as $\delta = \sqrt{\max(x)/\min(x)}$. Having the geometric spread, we can derive a (symmetric) multiplicative error bound for all elements $x_i \in X$ as:

$$(1/\delta)q \leq x_i \leq \delta q$$

where q is the geometric mean (q-middle).

4.4. The Link between Q-Error and Plan Quality

One can not expect that cost functions give exactly the same results as the measured costs, especially since the measured costs are typically non-deterministic. It follows that an error metric is required in order to measure the deviation of the estimated from the measured costs.

The error metrics we use is the *q-error*. The definition of the q-error was given in the previous section (see Eq. 4.1). The q-error itself is well known [9, 17, 21, 32, 48], but so far has only been applied to measure cardinality estimation errors. We apply it to measure the error of cost functions and show that there is a direct link between the q-error and plan quality.

Let $\mathcal{C}(e)$ denote the result of some cost function applied to some algebraic expression e, and let $\mathcal{M}(e)$ denote the true measured costs (e.g., runtime). Then, according to our definition (cf. Eq. 4.1), the q-error of the cost function $\mathcal{C}(e)$ is

$$\text{q-error}(\mathcal{C}(e)) = \|\mathcal{M}(e)/\mathcal{C}(e)\|_Q \,.$$

Choosing the q-error as the error metrics of choice is well justified by the following theorem and its corollary, for which we need some preparation. Let $\mathcal{E} = \{e_1, \ldots, e_k\}$ denote a set of plans. This set could be, for example, a

set of plans equivalent to a given query and generated/explored by the plan generator. However, $\mathcal{E}$ can be an arbitrary set of plans, making the theorem and its corollary very general. Further, let e_{opt} in be the optimal plan in $\mathcal{E}$ for a query Q, minimizing $\mathcal{M}(e)$, and e_{best} the best plan in $\mathcal{E}$, minimizing $\mathcal{C}(e)$. We are now interested in the factor by which the true costs of e_{best} are larger than the true costs of the optimal plan e_{opt}. An upper bound for this factor is given in the following theorem.

Theorem 4.4.1. *If for all* $e_i \in \mathcal{E}$

$$\|\mathcal{C}(e_i)/\mathcal{M}(e_i)\|_Q \leq q$$

for some q*, then*

$$\|\mathcal{M}(e_{best})/\mathcal{M}(e_{opt})\|_Q \leq q^2$$

Consider the case where $\mathcal{E}$ contains all the plans for a given query. Then, Theorem 4.4.1 tells us that if our cost function is precise up to a factor of q, then the plan picked under this (erroneous) cost function is at most a factor of q^2 away from the optimal plan. Since q^2 grows fast, this gives us some incentive to minimize q.

In terms of the cardinality estimation error, it was shown in [48] that the theoretical upper bound for the plan quality is higher, a factor of q^4, given that the q-errors of the cardinality estimates are bounded by q. In line with these two theoretical findings are the experimental results of Leis et al. [38]. They observe that cardinality estimation errors have a much higher impact on plan quality than cost model errors.

An important corollary to the theorem is:

Corollary. *If for all* $e_i \in \mathcal{E}$

$$\|\mathcal{C}(e_i)/\mathcal{M}(e_i)\|_Q \leq q$$

for some q *and for all* $e_i \neq e_{opt}$

$$q < \sqrt{\|\mathcal{M}(e_i)/\mathcal{M}(e_{opt})\|_Q},$$

then

$$\mathcal{M}(e_{best}) = \mathcal{M}(e_{opt}).$$

Thus, if the q-error of $\mathcal{C}$ is small enough (here $\leq q$), then the best plan chosen has the same cost as the optimal plan. Hence, the plan generator will still pick the optimal plan despite of the error in the cost function. This corollary, thus gives us an additional incentive to keep the q-error of our cost functions as small as possible. We now present the proofs.

Proof of Theorem 4.4.1 Since under the cost function $\mathcal{C}$ the plan e_{best} is minimal, we must have

$$\mathcal{C}(e_{\text{best}}) \leq \mathcal{C}(e_{\text{opt}}),$$

and since under $\mathcal{M}$ the plan e_{opt} is minimal, we have

$$\mathcal{M}(e_{\text{opt}}) \leq \mathcal{M}(e_{\text{best}}).$$

Since for all plans e we have $\|\mathcal{M}(e)/\mathcal{C}(e)\|_Q \leq q$, we can conclude that[3]

$$\begin{aligned} \mathcal{M}(e_{\text{best}}) &\leq q\mathcal{C}(e_{\text{best}}) \\ \mathcal{M}(e_{\text{opt}}) &\geq (1/q)\mathcal{C}(e_{\text{opt}}). \end{aligned}$$

Using all these inequalities, we can derive

$$\begin{aligned} \|\mathcal{M}(e_{\text{best}})/\mathcal{M}(e_{\text{opt}})\|_Q &\leq \frac{\mathcal{M}(e_{\text{best}})}{\mathcal{M}(e_{\text{opt}})} \\ &\leq \frac{q\mathcal{C}(e_{\text{best}})}{(1/q)\mathcal{C}(e_{\text{opt}})} \\ &\leq \frac{q\mathcal{C}(e_{\text{opt}})}{(1/q)\mathcal{C}(e_{\text{opt}})} \\ &\leq q^2 \end{aligned}$$

□

Proof of Cor. 4.4 Assume $\mathcal{M}(e_{\text{best}}) \neq \mathcal{M}(e_{\text{opt}})$. Then, by Theorem 4.4.1 we have the following contradiction:

$$\frac{\mathcal{M}(e_{\text{best}})}{\mathcal{M}(e_{\text{opt}})} \leq q^2 < \frac{\mathcal{M}(e_{\text{best}})}{\mathcal{M}(e_{\text{opt}})}$$

□

Note that the first inequality comes from Theorem 4.4.1 and the second one from Corollary 4.4.

4.5. Cost Model

Traditionally, query processing is performed in two separate phases: query optimization and query execution. In this approach, the query optimizer (QO) takes the input query and produces a query execution plan (QEP). Then, the query execution engine (QEE) evaluates the QEP to produce the query's result. The important link between the QO and the QEE is the cost model. The cost model consists of a set of cost functions, which model the resource consumption of the QEE for a given QEP.

As most QEEs are based on a physical algebra, the total costs of a QEP can be calculated by the sum of the costs of the physical operators contained therein, and the cost model needs to provide cost functions for all physical operators supported by the QEE.

On the other hand, the QO takes the cost model to evaluate different QEPs and to select the cheapest one among all those considered. To this end, it is important that the cost functions are as precise as possible. But what is the precise meaning of precise? What is needed is an error metrics that measures the deviation of the cost functions from the real costs measured by executing plans in the QEE. As there are plenty of metrics to be found in the literature, the question is which one is to be chosen for the purpose of query processing?

[3] $\forall x > 0 \;\; \|x\|_Q \leq q \Longrightarrow 1/q \leq x \leq q$

Attribute	Content
A	Ascending integers from 0 to n
B	Descending integers from n to 0
C	Integers randomly uniformly distributed in $[0, n]$
D	Ascending integers from 0 to n
E	Descending integers from n to 0
F	Integers randomly uniformly distributed in $[0, n]$
G	Ascending integers from 0 to n
H	Descending integers from n to 0

Table 4.2.: Attributes and their contents for the test relation R

We answered this question in Section 4.4 by providing a theorem that directly links cost function errors to plan quality. Since cost estimation errors have a profound negative influence on plan quality, it is important that the QEE allows for smooth and precise cost functions. In Chapter 3, we gave an example of a bad QEE to illustrate this point. Thus, the QO and the QEE very much depend on each other.

Cost functions in SystemTx are mostly linear combinations of linear components. Some of them contain branch misprediction costs as a non-linear component. In any case, the cost functions contain parameters that must be filled in. This process is called *calibration*, and it depends on the hardware. Calibration in our system is automated and proceeds as follows. A relation R with 9 different attributes and cardinalities is created. Initially, the cardinality of R is set to 2^{12} and then it is incremented in stepwise fashion up to 2^{28}. The attributes of R and their contents are summarized in Table 4.2, where n denotes the cardinality of R. Integers used for populating the relation R are of size 4 bytes, however, the same process can be applied for other types too.

For each size of R, three different plans are executed: (1) simple scans, (2) scans followed by a map operator with memory accesses, and (3) scans followed by a map operator and then by a selection operator. These plans correspond to plans a-c in Fig. 4.1. The selection operator in SystemTx depends on the values generated by the map operator, hence there is always a map operator preceding a selection operator. Since these plans are incrementally more complex, it is easy to extract the costs of a single operator from the measurements. For each operator, the extracted measurements are then approximated, using the cost functions. More details on the cost functions and their approximation are given in the following subsections.

As mentioned in the introduction of this chapter, our goal is to minimize the q-error; we do not use standard approximation techniques like linear regression, as they minimize the ℓ_2 error, which is not really useful in the context of query processing as shown in [48]. Instead, we apply the approximation techniques presented in [58], since they allow approximations that directly minimize the q-error.

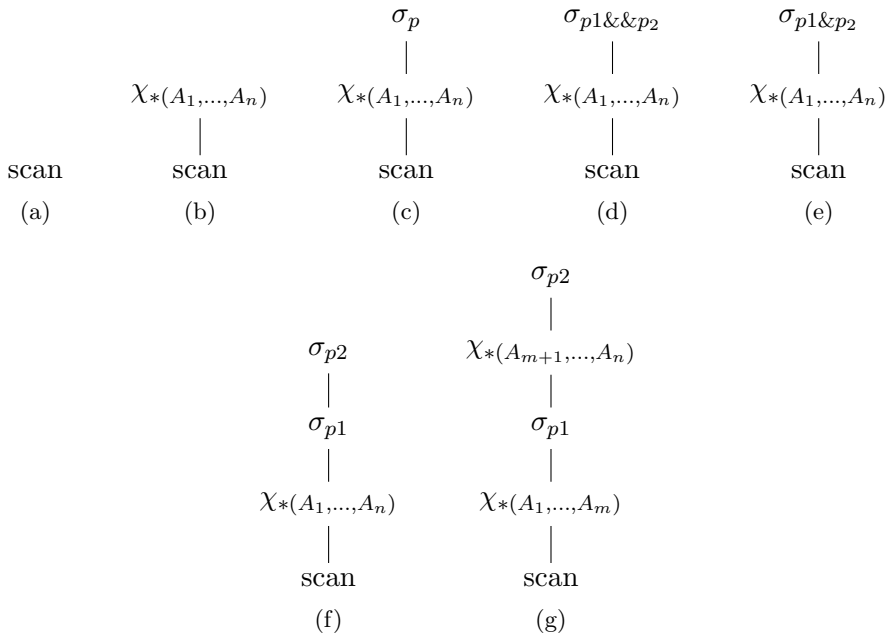


Figure 4.1.: Plan types

4.5.1. Cost Functions

For convenience, all notational details are summarized in Table 4.3, and all cost functions are presented in Table 4.4. Let us now briefly discuss the cost functions.

The scan and the map operator both exhibit linear costs, and their cost functions are thus rather simple (see Table 4.4). These cost equations can be derived by looking at the implementation details of each operator (such details for SystemTx are shown in Section 3.1). For example, the scan operator depends on the relation size $|R|$ as well as constants, e.g., cost of incrementing column iterator(s) and tuple pipelining:

```
TX_Scan::run() {
  for(i=0; i<|R|; ++i) {
    t.rid=i; t.ap++; t.bp++; t.cp++ ...
    consumer.step(t);
  }
}
```

In the pseudocode above, the column pointers are incremented inside the for loop, `t.ap++; t.bp++; t.cp++ ...`, whereas tuple pipelining is achieved by calling the step function on the consumer: `consumer.step(t)`.

In similar fashion, the map operator depends on the number of input tuples $|e|$ and the dereferenciation costs ($deref$) in addition to its constants (processing input/output tuples):

```
Tx_MAP1::step(t) {
```

Notation	Description
R	relation
$A_i, B_i, \ldots$	attributes, with and without index
$\mathcal{A}$	set of attributes
$\chi_{*(\mathcal{A})}$	map operator accessing $\mathcal{A}$
a_χ, b_χ	constants for map operator
$deref(d)$	costs of dereferencing d columns
p_i	predicates
s_i, sel(p_i)	selectivities for predicates
e	some algebraic expression (plan)
a_s, b_s	constants for scan operator
a_{in}, a_{out}	constants for processing input/output tuples
$\mathcal{B}(s)$	branch misprediction cost for selectivity s
$\mathcal{C}(e)$	cost function applied to e, estimated runtime

Table 4.3.: Notation

$$
\begin{aligned}
\mathcal{C}(scan(R)) &= |R| * a_s + b_s \\
\mathcal{C}(\chi_{*(\mathcal{A})}(e)) &= |e| * (deref(1,n) + a_\chi) + b_\chi \\
\mathcal{C}(p_1 \& p_2) &= \mathcal{C}(p_1) + \mathcal{C}(p_2) + \mathcal{C}(\&) \\
\mathcal{C}(p_1 \&\& p_2) &= \mathcal{C}(p_1) + \mathcal{B}(s_1) + s_1\mathcal{C}(p_2) \\
\mathcal{C}(\sigma_p(e)) &= |e| * (\mathcal{C}(p) + \mathcal{B}(sel(p)) + a_{in} + sel(p) * a_{out}) \\
\mathcal{C}(\sigma_p^{\pm}(e)) &= |e| * (\mathcal{C}(p) + \mathcal{B}(sel(p)) + a_{in} + a_{out})
\end{aligned}
$$

Table 4.4.: Cost functions

```
  t.A = R.A[t.rid];
  t.B = R.B[t.rid];
  t.C = R.C[t.rid];
  ...
  consumer.step(t);
}

Tx_MAP2::step(t) {
  t.A = *(t.ap);
  t.B = *(t.bp);
  t.C = *(t.cp);
  ...
  consumer.step(t);
}
```

In the pseudocode above, the dereferenciation is achieved by either accessing column values based on the rid variable, e.g., `t.A = R.A[t.rid];`, or by dereferencing column pointers, e.g., `t.A = *(t.ap);`.

In general, the dereferenciation costs can be replaced by general expression evaluation costs, especially if expensive function calls occur.

As shown in Section 2.2.2, a conjunction $p_1 \wedge p_2$ of predicates can be evaluated by expressions either of the form $p_1 \texttt{\&\&} p_2$ or of the form $p_1 \texttt{\&} p_2$, explaining the cost functions given in Table 4.4 for both of these expressions.

The cost function of the selection (σ) and bypass selection operator ($\sigma^{\pm}$) is a linear combination of linear and non-linear components. The non-linear component ($\mathcal{B}$) accounts for branch misprediction costs. For older database systems that still use an algebra that by tuple passing have an overhead, the scan together with the map and the selection operator can be merged into one operator; the cost of this new operator is then the sum of the cost of the scan, the map and the selection operator.

The bypass selection operator and the regular selection operator exhibit a small difference on their cost functions. That is, tuples flowing into the bypass selection operator are split into true or false stream depending on the outcome of the predicate:

```
Tx_Select_BYP::step(t) {
  if(p(t))
    consumer_true.step(t);
  else
    consumer_false.step(t);
}
```

In contrast, tuples flowing into the ordinary selection operator take only one stream (the true stream) if they satisfy the selection predicate, otherwise they are filtered out:

```
Tx_Select::step(t) {
  if(p(t)) consumer.step(t);
}
```

The bypass selection operator will come into play when we optimize Boolean expressions which contain disjunctions. More details on this topic are given in Chapter 6.

4.5.2. Memory Access Costs

Measuring memory access costs amounts to measuring the costs of our map operator $\chi_{*(A_1,A_2,\ldots,A_k)}$, for some attributes (i.e., columns) $A_1, A_2, \ldots, A_k$.

In SystemTx, there exist two ways of dereferencing column values, based on row identifiers, or based on column pointers. The pseudo code for both methods are shown in the previous section and Section 3.1.

The costs of the map operator clearly depend on the column access/dereferenciation costs. We measure the costs of the dereference operator by measuring the costs for plans shown in Figure 4.2. By subtracting the cost of the scan op-

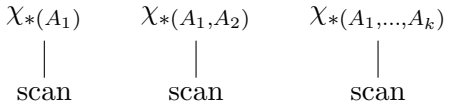


Figure 4.2.: Plan types for measuring the costs of the dereference operator

erator, we precisely capture the cost of the dereference operator. After we have isolated the costs for the dereference operator, we approximate them by taking the q-middle(x), where x denotes the dereference costs. For the definition of q-middle, see Table 4.1. That is, we use a single constant for each number of simultaneously accessed columns.

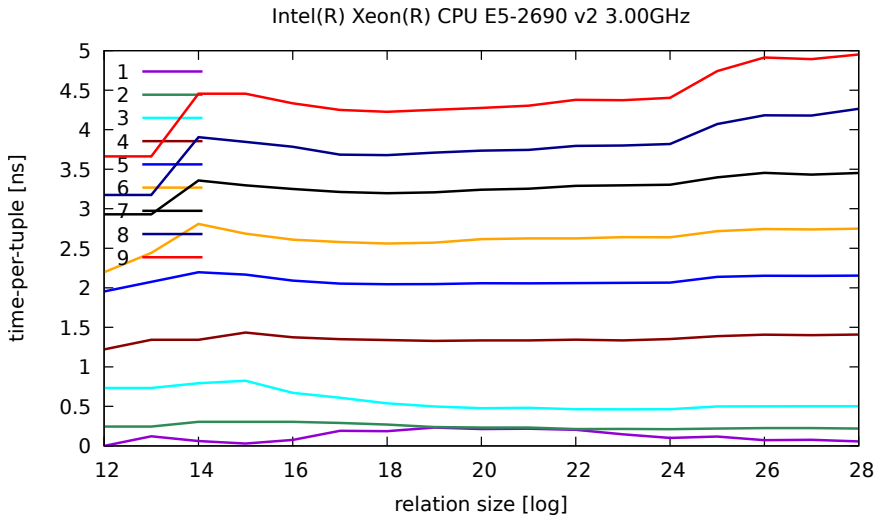


Figure 4.3.: SystemTx: column access costs

The run-times for a number of plans dereferencing up to 9 different columns are shown in Figure 4.3. The q-errors for all the plans and database sizes depicted in Figure 4.3 are shown in Figure 4.4. Note that we report the max

q-error for *all* database sizes ($[2^{12}, 2^{28}]$) for up to 9 simultaneous column dereferences at the time. The max q-error is very small for all the plans. In the

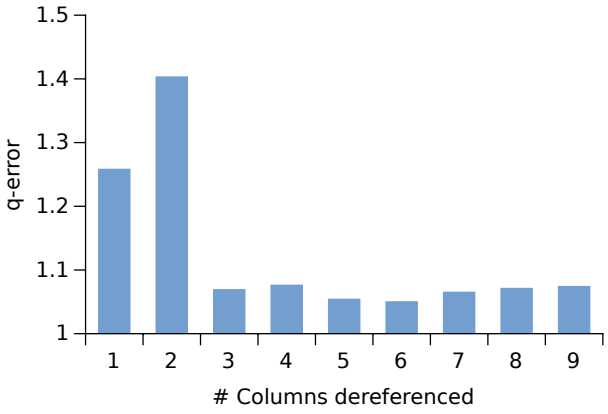


Figure 4.4.: Q-error of dereferenciation

worst case, for a plan dereferencing 2 columns at the same time, it can be off from the true costs by a maximum factor of 1.4. When the number of column accesses is greater than 2, the q-error drops below 1.1. The reason fur such a behavior is the cache size. For small relation sizes, the dereference operation are extremely fast as the columns fit into the cache. As a result, it gets hard to accurately measure the dereference operations, thus, the q-error is higher. For larger relation sizes, however, the dereference costs are more streamlined (and take longer) allowing for more accurate measurements, and hence, the q-error is lower.

4.5.3. Branch Misprediction Costs

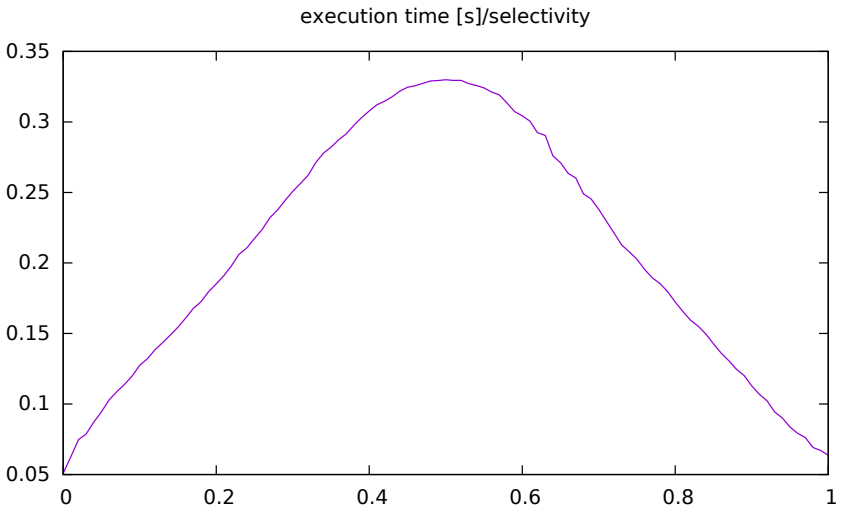


Figure 4.5.: Execution time of a selection operator

Fig. 4.5 shows the execution time of a simple scan over a column A, with a cardinality 2^{24} and with a selection predicate $A < \theta$ for varying θ and thus selectivities. The main reason for this hill shape is the branch misprediction penalty. Modern CPUs are very good at predicting branches when they are taken nearly always or never. The worst performance occurs at the selectivity of 0.5. At such selectivity, each branch outcome is taken with a probability of 0.5, thus making it hard for the CPU to predict it. The negative effects of the branch misprediction in the pipelined CPU execution model have been illustrated in Section 2.2.2.

In order to extract the branch misprediction cost from the execution time of a selection operator (σ), we proceed as follows. Recall the cost formula for the selection operator defined in Section 4.5.1:

$$\mathcal{C}(\sigma_p(e)) = |e| * (\mathcal{C}(p) + \mathcal{B}(sel(p)) + a_{in} + sel(p) * a_{out}).$$

For a selection over an attribute A belonging to some relation R, we have:

$$\mathcal{C}(\sigma_p(A)) = n * (a_{in} + \mathcal{C}(p)) + n * s * a_{out} + n * \mathcal{B}(s), \tag{4.2}$$

where n denotes the input cardinality (i.e., $n = |e|$), and $s = sel(p)$. Let us denote the measured cost for a given selectivity s by $\mathcal{M}(s)$. Then, Eq. (4.2) becomes

$$\mathcal{M}(s) = n * (a_{in} + \mathcal{C}(p)) + n * s * a_{out} + n * \mathcal{B}(s). \tag{4.3}$$

For selectivity 0,

$$\mathcal{M}(0) = n * (a_{in} + \mathcal{C}(p)),$$

and for selectivity 1,

$$\mathcal{M}(1) = \mathcal{M}(0) + n * a_{out} \ ,$$

and thus

$$a_{out} = \frac{\mathcal{M}(1) - \mathcal{M}(0)}{n}$$

Using these equations, we derive from Eq. (4.3)

$$\mathcal{M}(s) = \mathcal{M}(0) + s * (\mathcal{M}(1) - \mathcal{M}(0)) + n * \mathcal{B}(s), \tag{4.4}$$

and thus the branch misprediction cost for a given selectivity s is:

$$\mathcal{B}(s) = (\mathcal{M}(s) - \mathcal{M}(0) - s * (\mathcal{M}(1) - \mathcal{M}(0)))/n. \tag{4.5}$$

The branch misprediction can be very well approximated under the q-error [48] by a polynomial of degree 4, yielding a very low q-error: 1.08. The branch misprediction can also be well approximated by a cheaper piecewise approximation function:

$$\mathcal{B}(s) := \begin{cases} 6.264 * s + 0.0031 & s < 0.4 \\ -27.17 * s^2 + 26.88 * s - 3.96 & 0.4 \leq s \leq 0.6 \\ -6.065 * s + 6.065 & 0.6 < s \end{cases}$$

which yields a q-error of only 1.03. Note that the selectivity boundaries can be automatically derived using binary search.

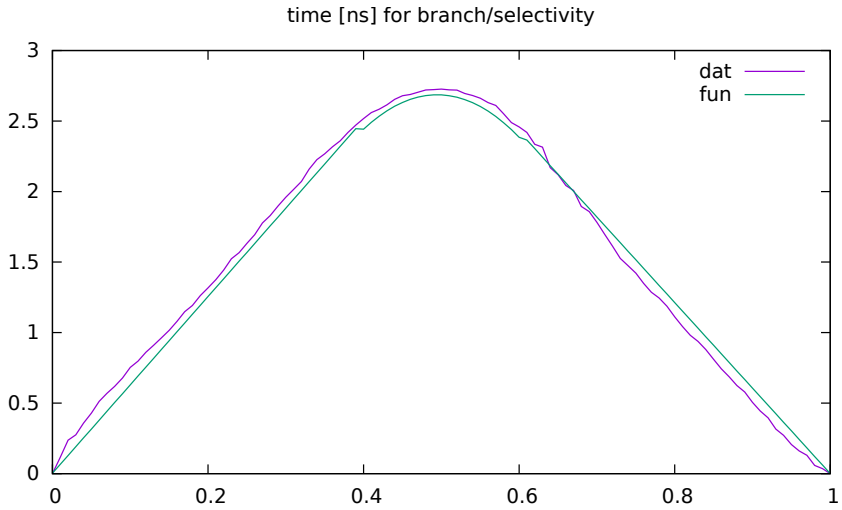


Figure 4.6.: Approximation of branch misprediction penalty

The results of approximation of the branch misprediction penalty using our piecewise approximation function have been depicted in Fig. 4.6, where "dat" denotes the actual branch misprediction penalty, whereas "fun" our approximation function. The plotted graph confirms also visually that our piecewise approximation function approximates indeed very closely the branch misprediction penalty.

4.6. Approximation of Cost Functions

Our goal is to approximate the cost functions of the operators build in SystemTx. These cost functions have been shown in Section 4.5.1. We have shown earlier in this chapter the benefits of using q-error and its theoretical implications, therefore, we are interested to approximate our cost functions with the aim of minimizing the multiplicative q-error. In this chapter, we provide some details on our approximation framework.

We are in particular interested into finding parameter values, so that we can find the *best fit* of experimental data. We have more experiments than unknowns, associated with experimental errors, therefore not a single model (function) will usually fit exactly the data, regardless of the parameter choice. Hence, we are interested in finding parameters that give us the best fit into the data that we get from experiments.

Let us consider a function $f(s)$ which given a selectivity $s \in [0, 1]$ as input, returns the run-time, for some operator. The function f can be represented by the following expression

$$\hat{f} := \sum_{i=1}^{n} c_i \phi_i(x)$$

where $\hat{f}$ is a linear combination of functions ϕ_i. The latter can be polynomial

or a linear function. Coefficients are denoted by c, where $c \in \mathbb{R}$. This equation defines $\hat{f}$ for all the points we wish to estimate. The argument x can be a scalar $x \in \mathbb{R}$, or a vector $x \in \mathbb{R}^n$. Finding the coefficients c constitutes our approximation problem.

Estimation of a particular point $\hat{t}_i$ is derived from $\hat{f}$:

$$\hat{t}_i := \hat{f}(s_i) = \sum_{i=1}^{n} c_i \phi_i(s_i)$$

For practical reasons, we represent the approximation problem in terms of vectors and matrices. A matrix A which we call it the *design matrix*, is defined by its elements $a_{i,j} = \phi_j(x_i)$, for some points (x, y) that we are interested to approximate, thus

$$A_{m,n} = \begin{pmatrix} \phi_1(x_1) & \phi_2(x_1) & \cdots & \phi_n(x_i) \\ \phi_1(x_2) & \phi_2(x_3) & \cdots & \phi_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(x_m) & \phi_2(x_m) & \cdots & \phi_n(x_m) \end{pmatrix}$$

In order to make this concept clearer, lets assume that we are interested in approximating the points by a polynomial of degree $n-1$, whereby $\phi_i = x^{i-1}$. We can take a quadratic function ($n-1=2$), such as $\hat{f}(x) = c_1 + c_2 x + c_3 x^2$, the design matrix is then populated as follows

$$A_{m,n} = \begin{pmatrix} 1 & (x_1)^1 & (x_1)^2 \\ 1 & (x_2)^1 & (x_2)^2 \\ \vdots & \vdots & \vdots \\ 1 & (x_m)^1 & (x_m)^2 \end{pmatrix}$$

The goal is to find $\vec{c}$, such that deviation of $A\vec{c}$ from $\vec{y}$ is minimal. Knowing that $m > n$, that is, there are more rows than columns (more equations than unknowns), the system is overdetermined. The likelihood that a solution to a such system exists is very small, i.e., the column space of A does not contain $\vec{y}$. *Therefore, we need to find an approximation with the smallest possible deviation from* $\vec{y}$. The deviation could as well be zero (although highly unlikely in overdetermined systems), thus

$$A\vec{c} = \vec{y}$$

In order to measure the metric distance (deviation) between real-valued functions such as approximation function $\hat{f}$ and the real function f, we need to define the *norm*, as metric distances are based on norms. Norm is also know in literature as the length of a vector or its magnitude, denoted with double bars, e.g., the length of the vector w is denoted by $\|w\|$. Length of a vector or its magnitude is a similar concept as the absolute value of real numbers, e.g., $|\lambda|$, which denotes the magnitude of a scalar λ.

Definition 4.6.1. (norm) Let V be a linear space, and $\|.\| : V \to \mathbb{R}$, where $\|.\|$ is a real-valued function over elements of the space V such that

1. $\|w\| > 0, w \neq 0$, otherwise $\|w\| = 0 \Leftrightarrow w = 0$

2. $\|\lambda w\| = |\lambda| \, \|w\|$, $\lambda \in \mathbb{R}$
3. $\|w + v\| \leq \|w\| + \|v\|$

Then $\|.\|$ defines a *norm* on the vector space V

Definition 4.6.2. Linear spaces such as V with a norm defined are called *normed linear spaces.*

Various norms exist, known as *p-norms*, or ℓ_p and are denoted with $\|.\|_p$. The *p-norm* is a norm metric over $\mathbb{R}^n$ (or $\mathbb{C}^n$), where $\|.\|_p$ is defined as:

$$\|w\|_p = (\sum_{i=0}^{n} |w_i|^p)^{1/p}$$

for $1 \leq p < \infty$. The most common norms known are $\ell_1, \ell_2, \ell_\infty$, and according to the above definition, they are computed as:

$$\|w\|_1 = |w_1| + |w_2| + \ldots + |w_n|$$
$$\|w\|_2 = \sqrt{(w_1)^2 + (w_2)^2 + \ldots + (w_n)^2}$$

and for $p = \infty$

$$\|w\|_\infty = \lim_{p \to \infty} \sqrt[p]{\Sigma_{i=1}^{n} |w|^p} = \max_{1 \leq i \leq n} |w_i|$$

where w_i stands for the ith component of the vector w. The ℓ_2 norm is the most common norm, it is also known as the **Euclidean** norm. The ℓ_∞ is the *minimax* or the **Chebyshev** norm.

Definition 4.6.3. For a given point g, and a set Z, where Z belongs to a normed linear space V, a point of Z that has a minimal distance from g, we call it a *best approximation.* The problem of determining such points with minimal distance is called a *best approximation problem.*

The distance between two vectors w, v in $\mathbb{R}^n$ is defined as follows:

$$d_p(w, v) = \|w - v\|_p$$

for $1 \leq p \leq \infty$. The distance defines the error metric, more generally, for a real (or complex) numbers, where the sum of such numbers is finite $\Sigma_{i=1}^{\infty} |w_i|^p$, or for $p = \infty, \Sigma_{i=1}^{\infty} |w_i|$, the error metric more generally is defined as

$$(\sum_{i=1}^{\infty} |w_i - v_i|^p)^{1/p}$$

For our purposes, the vector v denotes the approximation (obtained from $\hat{f}$) whose error we measure. Our goal is to find the *best approximation* under a normed vector space ℓ_p, knowing that different norms can produce different vector ordering.

We take ℓ_q as the norm of choice, for its definition see Section 4.3. The reason why we have chosen ℓ_q and its theoretical implications were given in Section 4.4.

4.6.1. Application in SystemTx

Having defined our approximation framework, the next step is to approximate the cost functions defined in Section 4.5.1. That is, we approximate the costs each operator in SystemTx in order to be able to estimate the overall query evaluation costs for query plans that our query optimizer enumerates. Having achieved this goal, the costs of some query evaluation plan P is then estimated as a summation of the (estimated) costs of the individual operators it contains:

$$\mathcal{C}(P) = \sum_{\text{op} \in P} \mathcal{C}(\text{op}).$$

However, before we can approximate the cost functions, we need to obtain the data points that we are interested to approximate. The data points are obtained by running query plans of different types as shown in Figure 4.1, over different relation sizes and parameters (constants occurring within the predicates). We measure the execution costs of each operator individually for the entire selectivity range $[0, 1]$. Details on extraction of the costs of the individual operators in SystemTx were given in Section 4.5. Our measurements will yield the data points that we are interested in to approximated. For example, for approximating the branch misprediction component of the selection operator (σ)—after we have isolated the costs of branch misprediction component, as shown in Section 4.5.3—we obtain the pair of points (s_i, t_i), where s denotes the selectivity and t the execution time.

In the second, step we determine the degree of functions ϕ_i, in order to construct the design matrix, which finally allows us to obtain the coefficients $\vec{c}$. For the branch misprediction component, we can choose a polynomial of degree 4, or a piecewise function, as shown in Section 4.5.3. The goal is to find the best approximation, i.e., to minimize the distance $d(A\vec{c}, \vec{y})$. The problem of finding the best approximation for multidimensional polynomials is transformed to a SOCP problem. More details on this step are given in [46]. SOCP is a convex optimization problem, and we solve it by means of the MOSEK [49] library, which performs this step efficiently. The measurements of the accuracy of approximation of the cost functions in SystemTx are shown in the following section.

4.7. Cost Model Validation

In order to validate our cost model, we have compared the measured execution times of several plans (see Figure 4.1) with the execution times predicted by our cost model. The plan types shown in Figure 4.1 were chosen as they cover most of the cases, and all other plan types build on top of them. Every plan was executed for different relation sizes and plan parameters, i.e., constants occurring within the predicates. For cost model validation we have used the same relation R which was used for the calibration in Section 4.5.1. The schema of this relation has been summarized in Table 4.2. For the validation purpose of our cost functions we have used relation sizes starting from 2^{10} and up to 2^{28}. The q-error we report is the maximum over the measurements over all

the relation sizes, i.e., 2^{10} up to 2^{28}. Table 4.5 shows the maximum q-error we observed for each plan type shown in Figure 4.1.

Plan type	q-error
(a)	1.09
(b)	1.1
(c)	1.08
(d)	1.34
(e)	1.09
(f)	1.14
(g)	1.27

Table 4.5.: True vs. estimated costs

Table 4.5 confirms that our cost functions are very accurate, yielding a maximum q-error of 1.3. That is, in the worst case, the upper bound on deviation of our approximated cost functions from the true costs can be a factor of 1.3. Thus, we conclude that our cost model is precise enough to serve the query optimizer's objective.

5. Optimization of Conjunctive Predicates

Optimization of queries with conjunctive predicates for main memory databases is a challenging task. The traditional way of optimizing this class of queries relies on predicate ordering based on selectivities or ranks. However, the optimization of queries with conjunctive predicates is a much more challenging task, requiring a holistic approach in view of (1) an accurate cost model that is aware of CPU architectural characteristics such as branch (mis)prediction, (2) a storage layer, allowing for a streamlined query execution, (3) a common subexpression elimination technique, minimizing column access costs, and (4) an optimization algorithm able to pick the optimal plan even in presence of a small (bounded) estimation error. In this chapter, we present an optimization algorithm for conjunctive queries which embraces the holistic approach, and show its superiority experimentally.

Current approaches typically base their optimization algorithms on at least one of two assumptions: (1) the predicate selectivities are assumed to be independent, (2) the predicate costs are assumed to be constant. Our approach is not based on these assumptions, as they in general do not hold.

The contents of this chapter were published in [33].

5.1. Introduction

It is not uncommon in data warehouses that decision support queries involve a larger number of conjunctive selection predicates. Data warehouses are increasingly storing tables in denormalized form [31] and in main memory, with the goal of achieving better query response times. In such settings, joins and disk I/O operations are not considered any longer the main cost [31], instead, the evaluation of selection predicates has replaced them as the dominating cost factor [31].

In this chapter, we focus on optimizing the class of conjunctive selection predicates of the form

$$p_1 \wedge p_2 \wedge \ldots \wedge p_n$$

in the context of main-memory column stores.

Definition 5.1.1. Predicates in our context are atomic. In their simplest manifestation, the predicates p_i are of the form $A\ \theta\ c$, where A is a column, $\theta \in \{=, \neq, <, \leq, >, \geq\}$, and c is a *literal* taking values from the domain of the *column*. We also consider expensive predicates of type `A like '%text%'`, as well as predicates applying expensive (user defined) functions.

The goal is to devise an optimization algorithm that determines the optimal evaluation order of selection predicates given in a query. As it turns out, this task is not as easy as it seems due to the details now becoming prominent.

Currently, two main approaches to optimize conjunctive queries can be found in (commercial) DBMSs. The first, rather simplistic approach orders the predicates by *increasing* selectivity and ignores the predicate costs [60, 66]. The second approach [26] orders predicates in *increasing* order of their ranks, where the rank of a predicate takes into account selectivities as well as evaluation costs and is defined as follows [28]:

$$rank = \frac{s-1}{\mathrm{c}} \tag{5.1}$$

where s denotes the predicate's selectivity and c its per-tuple cost. Under this optimization scheme, predicates with low costs and selectivities are given priority. The optimality of this approach can be proven for cost functions that exhibit the adjacent sequence interchange (ASI) property [28]. The ASI-property itself requires that the *independence assumption* (IA) holds. That is, there are no correlations between any two selection predicates, and the combined selectivity of any subset of predicates can be calculated by multiplying the single selectivities of the predicates contained therein.

It is well-known that this assumption in general does not hold [11]. To see this, consider the beautiful example of Markl et al. [41]: `make = 'HONDA' and model = 'ACCORD'`, where we observe the following. If we evaluate `make = 'HONDA'` first, its selectivity equals the market share of `HONDA` in our car database. If we evaluate `model = 'ACCORD'` first and then evaluate `make = 'HONDA'`, its selectivity will go up to 1.0, as there are no other car manufactures producing a model named `'ACCORD'`. This demonstrates that selectivities are *not* independent. To make things worse, changing selectivities have an impact on costs. Branch misprediction costs are maximal around a selectivity of 0.5 (see Figure 4.5 in Section 4.5.3) and drop significantly if selectivities approach either 0 or 1. Since for inexpensive predicates like comparisons, the branch misprediction costs are much higher than the predicate evaluation costs, neglecting branch misprediction costs results in very high error margins. Summarizing, predicate selectivities cannot be assumed to be independent, nor predicate costs to be constant. On the other hand, all previous approaches (see Section 5.2) rely on the assumption of constant predicate costs (CC) or IA.

In case of $p_1 \equiv 0.49 \leq A$ and $p_2 \equiv A \leq 0.51$, the attribute access costs exceed the predicate evaluation costs by far. Since after the evaluation of p_1 the attribute A has already been accessed, there is no need to access it again for p_2 [1]. Thus, the costs of evaluating p_2 drop significantly, showing the importance of common subexpression elimination (CSE). Most approaches do not take CSE into account (see Section 5.2).

A conjunction $p_1 \wedge p_2$ of predicates can be evaluated by expressions either of the form p_1`&&`p_2, or of the form p_1`&`p_2. The merits of either evaluation method have been shown in Section 4.5.1.

The rest of the chapter is organized as follows. Section 5.2 presents the related work. Section 5.3 presents the optimization algorithm, and Section 5.4

shows the experimental results.

5.2. Related Work

A number of commercial systems order predicates in increasing order of their selectivity without consideration of their costs. A good example is Vectorwise [60, 66], a well-known column store geared towards analytical workloads.

A more serious approach is presented by Hellerstein et al. [26]. They propose a scheme for ordering expensive predicates in an optimal way. Predicates in their work are not limited to cheap predicate, but can include non-trivial user defined functions (UDFs) that are expensive to evaluate. To this end, predicates are ranked in ascending order of the ranking metric shown in the Eq. (5.1), in the introduction of this chapter. This ranking metric originates from join-ordering domain [28, 37]. Hellerstein et al. [26] conclude that sorting of expensive predicates according to the above ranking metric produces the optimal plan. However, this is true only under the independence assumption. We have already seen that this assumption does not hold. Moreover, they do not consider CSE and in addition, predicate costs are assumed to be constant, i.e., they rely on the CC assumption, too.

Kemper et al. [36] consider optimizing boolean expressions in object databases by means of a heuristic based on Boolean difference calculus. Their algorithm is not limited to conjunctive case, it can handle disjunctions too. We consider disjunctions in Chapter 6, and compare this heuristic against our algorithms. For now, it suffices to know that the optimization algorithm in [36] assumes both CC and IA. Moreover, CSE is not considered.

Ross in [56] considers the optimization of conjunction of simple atomic predicates over arrays residing in main-memory with the goal of optimizing the branch misprediction costs. In particular, Ross [56] studies in detail the effect of conditional branches on plan quality and presents an algorithm which optimizes the branch misprediction penalty by cleverly connecting conjuncts with branching-and `&&`, and logical-and `&`. However, his algorithm does not consider CSE, and further, it relies on the IA. This, in turn, leaves a large optimization potential unharvested and calls for a new optimization algorithm that abandons both IA, CC and supports CSE.

The optimization algorithm shown in [56] has a time complexity of $O(4^n)$, for n atomic predicates. In contrast, the algorithm presented in this chapter has a much lower time complexity of $O(n\,2^n)$, as shown in Section 5.3, while it does not rely on the IA or CC, and, in addition, it supports CSE. In contrast to our work, the work in [56] does not provide *error bounds*.

The work by Munagala et al. [50] considers ordering of selection predicates by adopting approximation algorithms such as the set cover problem algorithm, coined *pipelined set cover*. The authors of [50] provide two approximation algorithms, an algorithm which is based on a greedy, and another based on a local-search heuristics. Their cost function simply counts the number of elements that each set covers, where, in turn, each set is mapped to an operator evaluating a selection predicate. Considering only the number of elements pro-

	Assumes		Supports	
	IA	CC	CSE	(&&),(&)
Kemper et al.[36]	Yes	Yes	No	No
Hellerstein et al.[26]	Yes	Yes	No	No
Ross [56]	Yes	No	No	Yes
Munagala et al.[50]	Yes	Yes	No	No
Neumann et al.[53]	Yes	Yes	Yes	No
here	No	No	Yes	Yes

Table 5.1.: Overview of related work

cessed does not provide an accurate cost function. Furthermore, this work relies on both constant predicate costs and the independence assumption.

Their work by Neumann et al. [53] focuses on identifying and eliminating common subexpressions involving expensive user defined function calls, and show that the problem of finding an optimal ordering of predicates while considering CSE is NP-hard. Their work is based on both constant predicate costs and the independence assumption.

The related work is summarized in Table 5.1, where we show the assumptions they make, the support of CSE, as well as the support of branching (&&) vs. non-branching (&) code.

5.3. The DPSel Optimization Algorithm for Conjunctive Predicates

In this section, we present an optimization algorithm coined DPSel. DPSel is responsible for producing query plans for evaluating conjunctions of selection predicates. It is based on dynamic programming. Figure 5.2 shows its pseudocode. DPSel attains the optimum in terms of plan quality.

DP algorithms generate solutions in a bottom-up fashion by combining solutions of subproblems [16]. That is, DP algorithms can be applied for problems that exhibit an *optimal substructure*. DP algorithms avoid recomputation of solutions for recurring subproblems by storing the solutions in a DP table; whenever the same subproblem recurs, its solution is fetched from the DP table instead of being recomputed each time it recurs, this way saving the recomputation costs.

DPSel accepts as input an expression with an arbitrary number of selection predicates connected conjunctively. Further, selectivities must be provided for each subset of the predicates occurring in the conjunction. These can be calculated beforehand, using the method of entropy maximization [42], or via the efficient sampling method shown in Chapter 8. In addition, the cost model presented in Chapter 4 is utilized to calculate the estimated plan costs. The output of DPSel is the best query evaluation plan, i.e., a plan with the lowest estimated execution cost. Thereby, DPSel relies on neither the IA nor the

BuildPlans(p, e)

```
    // Input: a selection predicate p
              an expression e (partial plan)
    // Output: plan container B
1   X_e = ∪_{p_i ∈ e} X_{p_i}
2   X_{p|e} = X_p \ X_e // outstanding maps
3   B = {σ_p(X_{p|e}(e))}
4   if e == σ_{p'}(X_{p|e}(e'))
5         B+=σ_{p'&p}(X_{p|e}(e'))
6         B+=σ_{p'&&p}(X_{p|e}(e'))
7   return B
```

Figure 5.1.: Pseudocode for BuildPlans

DPSel

```
    // Input: a set P = {p_0, . . . , p_{n−1}} of predicates
    // Output: an optimal plan
1   DP = an empty DP table, size → 2^n
2   DP[∅] = scan(R)
3   for each 0 ≤ i < 2^n − 1 ascending
4         P' = {p_k ∈ P | (⌊i/2^k⌋ mod 2) = 1}
5         for each p_j ∈ P \ P'
6               for each e_j ∈ BuildPlans(p_j, DP[P'])
7                     StoreSolution(e_j, P' ∪ {p_j}, DP)
8   return DP[P]
```

Figure 5.2.: Pseudocode for DPSel

StoreSolution(e, P, DP)

```
    // Input: an expression e
              a set of predicate(s) P
              a DP table
    // Output: none, affects DP
1   if DP[P] == NULL ∨ Cost(DP[P]) > Cost(e)
2         DP[P] = e
```

Figure 5.3.: Pseudocode for StoreSolution

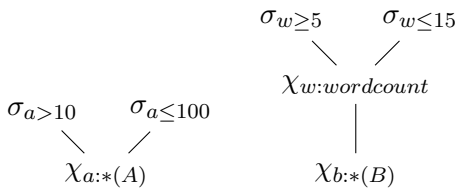


Figure 5.4.: Dependency graph for the example query

CC assumption. Moreover, it supports CSE, as well as branching-and(`&&`) and logical-and(`&`) predicate connectives.

The algorithm starts by initializing an empty DP table and storing a plan consisting of only the scan operator (cf. lines 1-2 in Figure 5.2). Selection operators evaluating predicates are built on top of this operator. The loop in line 3 iterates over all subsets P' of predicates P. The loop in line 5 iterates over the predicates in P which are not in P'. These are the new predicates that are not yet included in the existing partial plans stored in the DP table. Adding the new predicates to the existing (partial) plans is the responsibility of the BuildPlans procedure, shown in Figure 6.7. The BuildPlans procedure takes as an input a predicate and an existing partial plan.

A selection predicate depends on a certain set of map operators, thus forming the notion of the *dependency graph* [53]. For each operator that relies on values generated by some map operator, we draw an edge between that operator and the map operator on which it depends. For illustration purposes, consider the evaluation of the following query:

$$A > 10 \wedge A \leq 100 \wedge 5 \leq wordcount(B) \wedge wordcount(B) \leq 15$$

over some relation `R(A:int, B:text)`. Its dependency graph is shown in Figure 5.4. The UDF `wordcount` returns the word count of its input parameter, and it expects that the input parameter contains text. To this end, we are interested in finding all tuples which have for the attribute A their values in range of $(10, 100]$, and have a word count between 5 and 15 for the attribute values of B.

Selections involving attribute values of A depend on the map operator which generates the attribute values of A, whereas the selections involving values of the `wordcount` depend on the map operator which generates the values of the `wordcount`. The `wordcount` itself depends on the map operator generating attribute values of B, respectively. The attribute values of A in the above predicate are needed in two places, that is, there is a common subexpression. However, we can use only a single map operator generating the values of A, instead of two, this way eliminating the common subexpression. The same applies for the UDF function call `wordcount`, which is also needed in two places. UDF function calls can be much more expensive to evaluate than column dereference operations, therefore CSE is of crucial importance when searching for the optimal plan.

In the procedure BuildPlans, the set of dependencies that each input predicate p depends on, as well as CSE, are taken care of in lines 1,2. For the sequence

of selections in the partial plan e, their already executed map dependencies are denoted by

$$X_e = \cup_{p_i \in e} X_{p_i},$$

whereas the map dependencies of the input predicate p, which are still to be executed, are denoted by

$$X_{p|e} = X_p \setminus X_e.$$

After the map operators and CSE are taken care of, three different (logically equivalent) plans are created: 1) the input predicate is evaluated by a standalone selection operator added on top of the input plan, 2) the predicate is connected by the logical-and ('`&`') connection to the predicate(s) evaluated by the top selection operator in the input plan, and 3) the predicate is connected in a similar fashion as in 2), but by using the branching-and ('`&&`') connection instead of the logical-and. Plans of type (2) and (3) only make sense when the top operator of the existing partial plan e is a selection operator. This check is made in line 4 of the procedure BuildPlans. The newly constructed plans are returned to the main method. The main method (line 7) passes these plans to the StoreSolution procedure (see Figure 6.8), which in turn stores the *dominating* plan (the plan with the lowest cost) in the DP table, and other plans are *pruned*. Finally, the algorithm returns the best plan with the optimal cost for evaluating the given set P of selection predicates. The time complexity of DPSel is $O(n\,2^n)$, for n predicates in a conjunctive query. The algorithm has to interate over all the subsets of predicates, hence the term 2^n (cf. line 3). The term n comes from line 5; the algorithm iterates over all predicates in P which are not in P'.

Subset enumeration (cf. lines 3 - 4 in Figure 5.2) can be very efficiently computed by means of bitvectors. In bitvector representation, the numbers from 0 to $2^n - 1$, incremented by 1 represent all subsets of P. Such increments by 1 are in line with the DP strategy: for each subset P', all subsets of P' are generated before P' itself. Further, increments by 1 is a very fast machine operation requiring only one CPU instruction.

In our implementation, bitvectors are of integral types $uint\_32t$ or $uint64\_t$. That is, their width is limit to a word size (32 or 64 bits depending on the architecture). Each bit position i in bitvector represents a predicate p_i. An example illustrating a set of predicates encoded in a bitvector representation has been depicted in Figure 5.5. In bitvector representation, one can very efficiently iterate over the bits set by means of assembler instructions, e.g., `_bit_scan_forward(`bv`)` [29] and `_bit_scan_reverse(`bv`)` [29]. The first instruction (`_bit_scan_forward(`bv`)`) returns the index of the least significant bit set to 1, whereas the second instruction (`_bit_scan_reverse(`bv`)`) does the opposite, it namely returns the index of the most significant bit set to 1.

Map dependencies in SystemTx are also stored in a bitvector, thus the computation of X_e and $X_{p|e}$ can be done very efficiently by means of bitwise operators (e.g., OR, XOR) — requiring only few CPU operations.

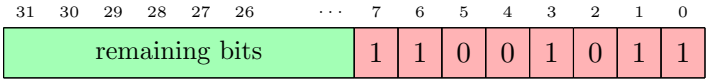


Figure 5.5.: A bitvector of integral type $uint32\_t$ representing a set of predicates $P = \{1, 2, 4, 7, 8\}$

5.4. Evaluation

The evaluation of predicates in data warehouses has become the major bottleneck for decision support queries [31]. We show in this section that there remains a large optimization potential unharvested by other commonly used heuristics based optimization algorithms in RDBMSs. For the experimental evaluation of our optimization algorithm DPSel, we compared it against two widely used heuristics algorithms.

In some commercial systems, predicates are simply ordered in increasing order of their selectivities. One example of such a system is Vectorwise [60, 66]. We term the algorithm that orders predicates in ascending order of their selectivities as Sel. Other systems order predicates in increasing order of their ranks, where ranks are computed according to Eq. (5.1). We call this algorithm Rank.

In this section, we are interested in answering three main questions:

1. what is the loss of plan quality if we apply Sel or Rank compared to DPSel,
2. what is the cost of applying DPSel instead of Sel or Rank, and
3. what is the loss on plan quality in the presence of cardinality estimation errors.

For testing qualities of plans produced by DPSel vs. the other two heuristics algorithms, we have performed two sets of experiments. For the first set of experiments we used predicates with varying costs (general case), whereas for the second set of experiments, we have used inexpensive predicates with equal costs (special case). We have enriched the experimental evaluation by running additional experiments using the TPC-H [18] and the Forest [14] dataset.

In order to set up the selectivities needed if we abandon the IA, we generated a pool consisting of 100 different predicates joint selectivities for queries containing up to 10 conjunctive atomic predicates. That is, for each combination of predicates and their subsets, 100 different joint selectivities were available.

Selectivities for single predicates p_i and pairs $(p_i \wedge p_j) \forall i, j$ were generated randomly, uniformly distributed in the range $[0, 1]$. Their consistency was ensured by means of PDHGMp [47]. When randomly generating predicate selectivities, there can be inconsistencies, therefore we have used PDHGMp algorithm for generating consistent predicate selectivities. For the rest of predicates $\wedge_{i \in I} P_i, I \subseteq \{1, \ldots, n\}$, their joint selectivities were generated by the principle of *maximum entropy* (ME) [42].

We conclude the Experiments section with a comparison of the running times of the three algorithms. The experiments were run single-threaded, on a ma-

# subexpr.	Nr. of predicates							
	3		4		5		6	
	RANK	SEL	RANK	SEL	RANK	SEL	RANK	SEL
1	2.6	2.7	3.1	4.1	3.5	10.3	4.3	35.1
2	2.6	2.6	3.1	3.9	3.5	8.3	4.2	15.9
3	2.6	2.6	3.1	3.1	3.4	3.4	3.8	3.8

Table 5.2.: Relative optimization potential (in factors!) of DPSEL vs. RANK and SEL for the range of predicates 3-6.

# subexpr.	Nr. of predicates							
	7		8		9		10	
	RANK	SEL	RANK	SEL	RANK	SEL	RANK	SEL
1	5.1	38.4	5.8	64.3	6.4	80.5	6.9	110
2	5	21.3	5.7	29.6	6.3	35.2	7.2	42.7
3	4.3	4.3	4.8	4.8	5.3	5.3	5.7	5.7

Table 5.3.: Relative optimization potential (in factors!) of DPSEL vs. RANK and SEL for the range of predicates 7-10.

chine with Intel Xeon E5-2690 v2 3.00GHz processor. The machine had 120 GB of main memory, running a 64-bit linux operating system. The algorithms were implemented in `C++` and compiled with Intel's `icpc` compiler.

5.4.1. General Case

In this section, we show the results of the performance of DPSEL vs. the other two algorithms in terms of plan quality by using predicates with varying costs. Selection operators make only comparisons ($=, \neq, <, \leq, >, \geq$) over the values generated by subexpressions which they depend on, therefore, their cost was set to 1. The costs of the subexpressions that selection predicates depended on were generated randomly, uniformly distributed in the range [1,1000].

We ran three different sets of experiments, whereby in each experiment we have tested the algorithms starting with 3, and up to 10 predicates, and a pool containing in total 3 subexpressions. For each number of predicates, we ran the algorithms 100 times. For each run, different predicate joint selectivities were picked from the pool of joint predicate selectivities. For the first experiment, each predicate depended on values generated by a single subexpression. We assigned 1000 different random cost values to the subexpressions.

Since we were interested in finding the maximum optimization potential between DPSEL and the other two heuristics algorithms, we recorded the plans with the maximum cost difference from all the runs. We repeated the same experiment and we varied the number of subexpressions on the dependency graphs. That is, we performed two more experiments, where the dependency graph for each predicate contained two and three subexpressions, respectively.

The results of this experiment are shown in Table 5.2 and Table 5.3. As the plan costs varied greatly, the plan costs of SEL and RANK are given relative to

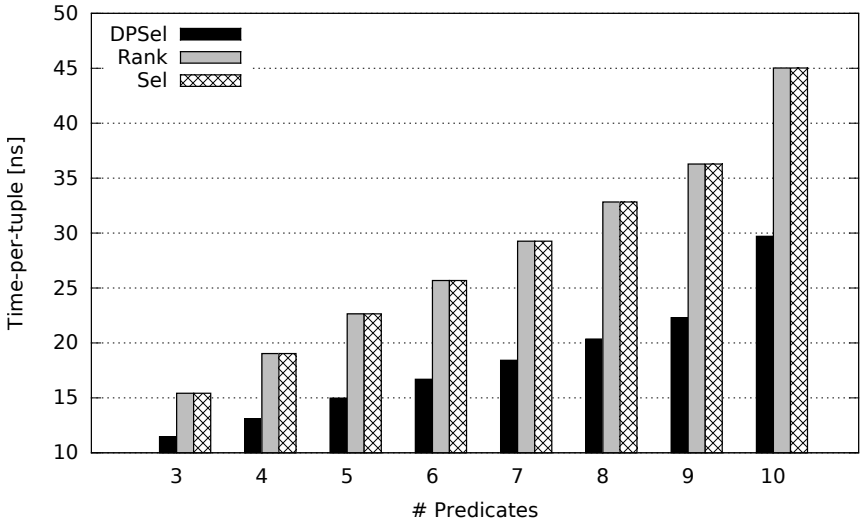


Figure 5.6.: Plan costs for inexpensive predicates sharing a single subexpression

DPSel. Thus, the tables contain the factors by which the plans produced by Sel and Rank are worse than the plans produced by DPSel.

For all the experiments, plans generated by DPSel outperformed by a large margin both heuristics based algorithms. Starting with 3 predicates, DPSel outperformed Rank and Sel by a factor greater than 2, for all sizes of dependency graphs. With the increase in the number of predicates, the gap on plan qualities increased as well such that for 10 predicates DPSel beats Sel by a whopping factor of 110, and Rank by a factor of 7.

Our experiments show that query optimizers relying on heuristics for optimization of conjunctive predicates produce plans that can be as far as a factor of 110 away from the optimum.

5.4.2. Special Case

In this section, we list our experimental findings of comparing the qualities of plans generated by DPSel and the other two heuristics (Rank, Sel) for inexpensive predicates with costs equal to 1. That is, we have limited the cost of the subexpressions to 1. As in the previous section, the cost of selection predicates was set to 1, as they perform only comparison operations ($=, \neq, <, \leq, >, \geq$) over the values generated by their respective subexpressions.

As in the previous subsection, we have tested the algorithms starting with three, and up to ten predicates. For each number of predicates, we ran the algorithms 100 times. For each run, a different predicates joint selectivity was picked from the pool of joint selectivities.

For the first experiment, all the predicates were assigned dependency graphs containing a single subexpression. The results of this experiment are shown in Figure 5.6. The y-axes shows the per-tuple cost in nanoseconds (ns), whereas the x-axes shows the number of predicates.

The algorithms Sel and Rank produced the same results, due to the fact that

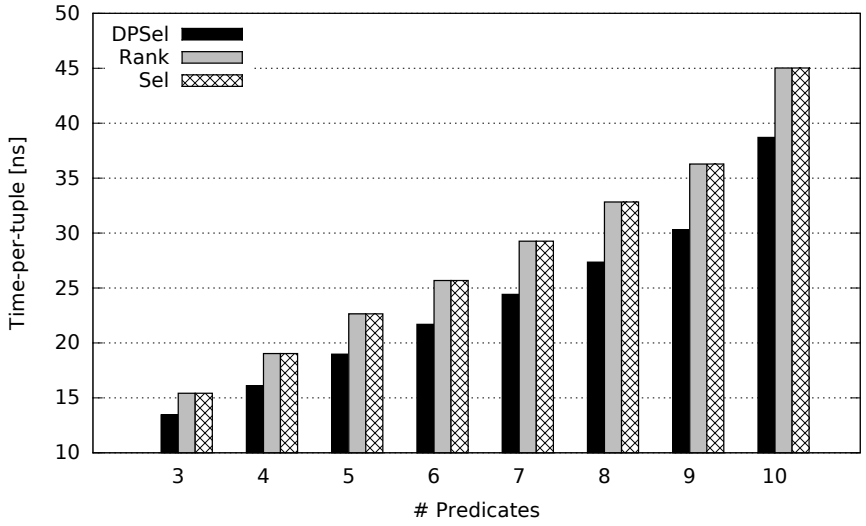


Figure 5.7.: Plan costs for inexpensive predicates, no shared subexpression

all predicate costs are equal, and thus can be safely neglected. For all the numbers of predicates, DPSel is the clear winner. Since all the predicates depended on one subexpression, DPSel applies CSE. In addition to CSE, DPSel also minimizes the branch misprediction costs. Whereas in the case of Rank and Sel, the subexpression is evaluated for each selection, as CSE is not considered there. In addition, the two heuristics do not minimize the branch misprediction penalty.

Starting with three predicates, DPSel produced plans that are a bit over 20% cheaper than those produced by Rank and Sel. With the increase in the number of predicates, the difference on plan quality produced by DPSel and the other two algorithms increased as well. For 10 predicates, the difference on plan qualities was as large as 40% in favor of DPSel. This is a large optimization potential, considering that we have tested the algorithms for inexpensive predicates.

We repeated the same experiment, but this time each selection depended on the values generated by one unique subexpression. That is, there were no shared subexpressions among selection operators. This way, we have eliminated the optimization potential that DPSel harvests by employing CSE. The results of this experiment are shown in Figure 5.7.

We observe that the costs of plans produced by DPSel are nevertheless lower than those produced by Rank and Sel. This time, though, DPSel produced better plans solely due to optimization of the branch misprediction penalty.

We conducted yet another experiment. This time, we generated a pool of 10 different subexpressions. Each selection predicate formed a dependency graph containing 3 different subexpressions chosen randomly from the pool of subexpressions. As in Section 5.4.1, 100 different dependency graphs were generated. As before, the algorithms were tested using 100 different predicates joint selectivities. There results of this experiment are shown in Figure 5.8.

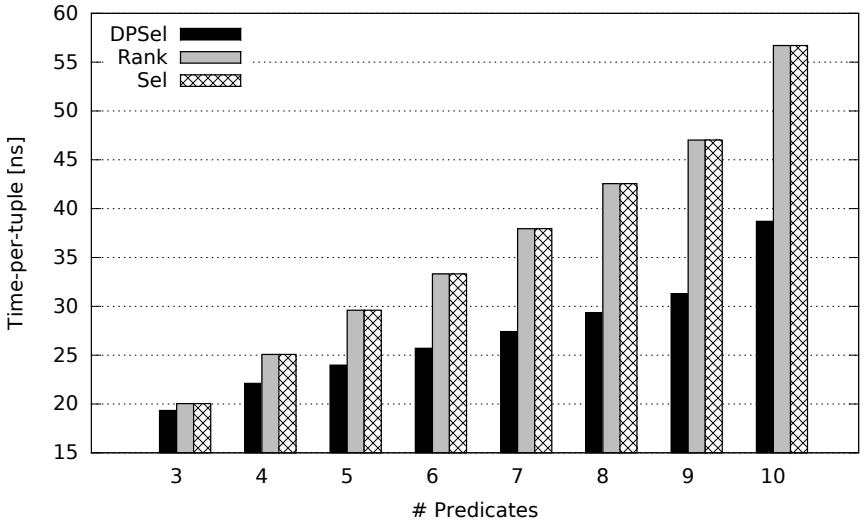


Figure 5.8.: Plan costs for inexpensive predicates depending on 3 different subexpressions

Algorithm	Est. plan cost (ns)	Evaluation cost (ns)
DPSel	8.49	8.99
Rank/Sel	12.09	12.59

Table 5.4.: DPSel vs. Rank and Sel over TPC-H dataset

We observe similar results to the case when a single subexpression was shared among all selections (see Figure 5.6). Despite the fact that we have only evaluated cheap predicates with fixed costs, DPSel produced plans that are over 40 % cheaper than the heuristics based algorithms. For all the number of predicates, DPSel consistently beats Rank and Sel.

5.4.3. TPC-H Dataset

This section presents the results of comparing DPSel vs. Rank and Sel by using the TPC-H [18] dataset. For the TPC-H dataset, we have used a query with three predicates over the `lineitem` table:

```
SELECT * FROM lineitem
WHERE orderkey <= 5889891
AND partkey <= 153588
AND suppkey <= 9960;
```

The `lineitem` table was generated using scaling factor (SF) 1, yielding a total of slightly over 6 million tuples. Since DPSel does not rely on the IA, the predicate selectivities for all subsets of predicates given in the query were computed a-priori by means of the sampling method shown in Chapter 8.

The results of this experiment are shown in Table 5.4. The middle column of the table shows the estimated plan costs in time-per-tuple for each algorithm, whereas the last column shows the actual measured plan costs by running the actual plans in SystemTx. Note that in Table 5.4, there is one entry for both Rank and Sel, as both heuristics produced the same plan due to the fact that predicate costs were equal for all the predicates.

The loss in plan quality of heuristics based algorithms relative to DPSel is a factor 1.4 or 40%. This is a huge gap considering that the predicates were cheap to evaluate, there were no common subexpressions, and the query contained only three predicates!

In addition to the gap on plan qualities, this experiment confirms that our cost model is extremely precise: the estimated plan costs differ from the true measured costs only after the decimal point.

5.4.4. Forest Dataset

In this section, we present the experimental evaluation of DPSel vs. Rank and Sel by using the Forest [14] dataset.

The materialized relation of the Forest dataset contains 54 attributes, and 581.012 tuples. This rather wide relation validates the importance of optimizing conjunctive queries.

For the Forest dataset, we used 4 cheap range predicates over different attributes of the Forest relation. That is, all predicates had equal costs. The predicates were of the type $c_1 \leq attr_i \leq c_2$, where c_1, c_2 denote integer constants.

We generated randomly 1 million queries over randomly selected attributes of the Forest relation with random predicate constants (i.e., c_1, c_2). The results of this experiment are shown in Table 5.5.

Algorithm	Equal costs	Varying costs
Rank/Sel	2.01	21.42

Table 5.5.: Relative optimization potential of DPSel vs. Rank and Sel over the Forest dataset

DPSel beats the other two heuristics algorithms by a factor of 2. An optimization potential of factor 2 is quite large, considering that predicates were cheap to evaluate, and the query contained only 4 predicates.

We have repeated the same experiment with the Forest dataset, but this time we assigned to subexpressions random costs uniformly distributed in the range [1,100]. As expected, DPSel beats the other two algorithms, this time by a large factor of 21 (cf. Table 5.5, third column).

5.4.5. Plan Quality Loss in Presence of Cardinality Estimation Errors

We cannot expect that a database system has detailed, and more importantly correct knowledge about the joint frequency distribution of attribute values for

f	Equal costs	Varying costs
2	2.27	3.03
3	2.66	5.03
4	3.14	6.97
5	3.3	8.64

Table 5.6.: The max q-error between e_{best} and e_{opt} for different f values

a relation of interest. In this section, we experimentally investigate the influence of estimation errors on the plan quality for conjunctive predicates.

In order to introduce a defined error, we have deliberately multiplied the true predicate selectivities with an error factor (f). The goal was to find the maximum deviation factor on the plan quality between e_{opt} and e_{best}, where e_{opt} denotes the optimal plan and e_{best} denotes the best plan picked under an erroneous cost function, i.e., a cost function which has to work with erroneous predicate selectivities.

For this experiment, we have used again the Forest [14] dataset, a set of eight predicates, and a pool containing $10k$ different predicate joint selectivities. All the predicate joint selectivities were multiplied by the error factor f. There were $1k$ different values picked randomly from the set $\{f, 1/f\}$, for all $f := \{2, 3, 4, 5\}$. For predicates with varying costs, 100 different values for subexpression costs were chosen, uniformly randomly distributed in the range $[1, 100]$. For predicates with equal costs, all subexpressions were assigned equal costs.

The maximum deviation ratio ($\mathcal{M}(e_{best})/\mathcal{M}(e_{opt})$) over all runs was recorded. Recall that $\mathcal{M}(e)$ denotes the true measured costs for some plan e.

The results of this experiment are shown in Table 5.6. In the light of theorem 4.4.1, the maximum deviation on plan costs between e_{best} and e_{opt} is surprisingly low. That is, the maximum deviation factor on the plan quality between e_{opt} and e_{best} remains well below f^2 for all values of f.

5.4.6. Runtime

In this section, we show the performance of DPSel against Rank and Sel in terms of their running times.

We measured the runtime performance of the three algorithms, starting with two and up to 10 predicates in total. The results of these measurements are shown in Figure 5.9. The y-axis denotes the runtime in milliseconds (ms), whereas the x-axis denotes the number of predicates that were fed to the algorithms.

Although DPSel has a complexity proportional to $O(n\,2^n)$ for n predicates, our experiment shows that its runtime for up to 10 predicates is nevertheless very low, under 0.6 milliseconds. Considering its optimization potential of factor 7 against Rank, and factor of 110 against Sel, the optimization time under 0.6 ms is certainly worth the price. Further, DPSel guarantees the optimum in terms of the plan quality.

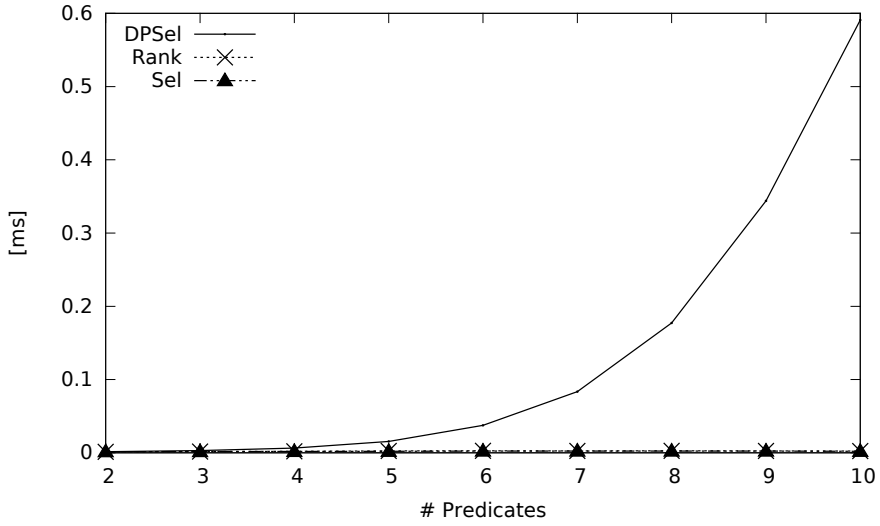


Figure 5.9.: The evaluation results of runtime performance

5.5. Conclusion

In this chapter, we presented an optimization algorithm for conjunctive queries that does not rely on assumptions like IA and CC. Furthermore, it takes CSE into account, while supporting logical-and(`&`) and branching-and(`&&`) for evaluating conjunctions. Experimentally, we showed that the loss in plan quality if relying in IA and CC can be as high as a factor of 100, compared to the optimal plan.

Since cost models play a crucial role in query optimization, we spent some pages not only to present a cost model, but also to argue that the q-error is the preferred metrics to measure the deviation of actual from estimated plan costs. This is due to a new theorem presented that directly links the q-error of a cost model to plan quality. To the best of our knowledge, this is the first time such a link has been proven for any error metric.

6. A Heuristic for Boolean Expressions

Optimization of Boolean expressions is a very challenging task which has been vastly neglected by the research community and commercial databases. In this chapter, we focus on the complex problem of optimization of Boolean expressions by means of the *bypass* processing technique. In bypass processing, selection operators split the input tuple stream into two disjoint output streams: the *true-stream* with tuples that satisfy the selection predicate and the *false-stream* with tuples that do not. Bypass processing is crucial in avoiding expensive predicates whenever the outcome of the query predicate can be determined by evaluating the less expensive ones.

We have already shown in Chapter 5 that for main memory databases, CPU architectural characteristics, such as the branch misprediction penalty, become a prominent cost factor which cannot be ignored. In Chapter 5, we have presented an optimization algorithm which optimizes queries containing only conjunctive predicates, whereas in this chapter we present an optimization algorithm which, besides conjunctions, also handles disjunctions. The optimization algorithm presented in this chapter takes into account the branch misprediction penalty, duplicate predicate elimination together with common subexpressions elimination.

The current literature relies on two assumptions: (1) predicate costs are assumed to be constant, and (2) predicate selectivities are assumed to be independent. Since both assumptions do not hold in practice, our approach is not based on any of them.

In this chapter, we present a heuristic for optimizing Boolean expressions whereas in the next chapter we present an optimization algorithm which guarantees the optimum in terms of plan quality. The contents of this chapter have been published in [34].

6.1. Introduction

The optimization of Boolean expressions containing predicates connected by the disjunction OR ($\vee$) is a topic which has been neglected by the database research community with the exception of a few publications. Instead, the focus was set on the optimization of conjunctive predicates, as disjunctive predicates are much more complex, and it was claimed that they do not occur often in practice. This might have been true before, however, with the advent of main memory database systems, this is changing. Main memory databases are especially prevalent in decision support systems, where relations are being increasingly stored in denormalized form due to the demands for high query response time. In such settings, joins are not considered any longer the main cost, instead,

selection predicates have taken this role [31]. A good example supporting this claim is query Q1 taken from a customer workload [31]:

```
A = a AND B = b AND C IN (c,d)
AND D = 0 AND E = 0 AND F <= 11
AND G >= 2 AND H IN (e,f)
AND (I IN (g,h) OR I > j).
```

Further, the authors in [10] show that data mining and data warehouse applications often generate queries with long and complex predicate expressions, requiring a query optimizer that generates efficient plans despite the complexity of predicate expressions. To support this argument, they traced a real database, containing 900 tables and 600MB of data. The trace contained 1931 queries, which were dominated by selections, and only few aggregate and join queries [10].

As we will show experimentally in this chapter, there is a vast optimization potential available for Boolean expressions, which we cannot afford to neglect any longer, if our aim is to build responsive main memory database systems.

The optimization of Boolean expressions in relational database management systems has been traditionally done by transforming Boolean expressions into either conjunctive normal form (CNF) [57] or disjunctive normal form (DNF) [30, 61].

In CNF optimization scheme, disjuncts within Boolean factors can be ordered optimally (locally) according to the equations in [25]. In DNF optimization scheme, in the same fashion as in the CNF scheme, conjunctive predicates within Boolean summands can be ordered optimally (locally) as shown in [25].

Optimality of the ordering methods shown in [25] can be proved only if the independence assumption (IA) holds. However, it is well known that this assumption does not hold (see Chapter 5). It was also shown in Chapter 5 that changing predicate selectivities impacts the predicate costs. Fig. 4.5 shows the execution time of a simple scan over a column A (containing 2^{24} entries) and a simple selection predicate $A > c$ for varying c, and thus selectivities. The bell-shaped curve is the result of the branch misprediction latency, which between selectivity 0 and 1 can be as high as factor 7. Thus, it cannot be ignored. The optimization algorithm presented in this work is branch misprediction aware. Summarizing, predicate selectivities cannot be assumed to be independent, nor predicate costs to be constant.

When optimizing conjunctive predicates, we have seen the importance of common subexpression elimination (cf. Chapter 5), the same applies for disjunctive predicates too. Most approaches do not take CSE into account (see Sec. 6.2). Further, in terms of disjunctive queries, duplicate predicates have a profound impact on the plan quality, which, if not correctly recognized by the optimizer, can lead to very poor plans. Consider the following example query $(p_1 \wedge p_2) \vee (p_1 \wedge p_3)$. Let us assume for the moment that the evaluation of predicate p_1 resulted in a *false* outcome. Regardless which Boolean summand (i.e. $p_1 \wedge p_2$ or $p_1 \wedge p_3$) was picked first, the second Boolean summand should not be evaluated at all, as the result of this query is determined to be false.

The algorithms presented in this chapter detects common duplicate predicates and reduces the query accordingly.

When optimizing Boolean expressions, the maximum theoretical optimization potential cannot be achieved by means of traditional plans. We can, however, fill this gap by means of *bypass processing* [12, 35]. Using this technique, selection operators split the tuple stream into two disjoint streams; the *false* streams with tuples that do not satisfy a predicate, and the *true* stream with predicates that do. The bypass processing technique is instrumental in avoiding expensive predicates altogether, whenever the outcome of the query can be determined by evaluating the cheap ones.

We summarize our contributions in this chapter as follows:

1. we present an optimization algorithm that, besides conjunctive predicates, can also handle disjunctive predicates. Our algorithm relies neither on the IA (independence assumption) nor the CC (constant predicate costs assumption),

2. our algorithm supports CSE (common subexpression elimination) together with the branch misprediction latency,

3. our algorithm generates bypass plans, this way filling the theoretical gap on the vast optimization potential which cannot be harvested by traditional plans,

4. our algorithm supports elimination of shared predicates among different boolean summands.

For in-memory database systems where queries are JIT compiled, e.g., HyPer [52], the overhead of explicitly creating the physical bypass operators does not exist. There, the system can generate the code for bypass operators as nested *if statements*, this way making the algorithm presented in this chapter very attractive for such systems.

In this chapter, we discuss the construction of bypass plans for in-memory column stores. However, the algorithm presented in this chapter does not depend on the storage layout, but only the cost functions depend on the storage layout; the algorithm can work with any cost function. The algorithm presented in this chapter, however, has few disadvantages. It does not guarantee the optimum in terms of plan quality, and further, it requires the queries in DNF. In the worst case, if the query is in CNF, the conversion to DNF yields a query which is exponentially blown-up in size. The algorithm presented in the next chapter does not suffer from these disadvantages.

The rest of the chapter is organized as follows. Section 6.2 contains the related work. In Section 6.3 we present the preliminaries for this chapter. In Section 6.4 we show different plan construction strategies for Boolean expressions. Section 6.5 presents the optimization algorithm. Section 6.6 shows the evaluation results.

6.2. Related Work

As stated in the introduction, the most prevalent way of optimizing disjunctive queries in commercial databases is based on either of the two normal forms: CNF (like System R [57]) or DNF [30, 61]. The work in [25] shows how to order both conjunctive ($\vee$) and ($\wedge$) disjunctive predicates optimally. In [25], a Boolean expression is represented as a tree of ($\vee$) and ($\wedge$) nodes, whereby the terminal nodes (leaves) represent the predicates in a given Boolean expression, e.g., attribute θ constant. Let $E_\wedge$ denote a ($\wedge$) node in the tree representation of the Boolean expression, and $E_\vee$ denote a ($\vee$) node, respectively. The work in [25] says that the ordering of children nodes of $E_\wedge$ according to the formula

$$\frac{c(x_1)}{1-s(x_1)} \leq \ldots \leq \frac{c(x_n)}{1-s(x_n)} \tag{6.1}$$

produces an optimal ordering, that is, the expected evaluation time of $E_\wedge$ is minimal. Note that $c(\cdot)$ and $s(\cdot)$ denote the cost and the selectivity of a predicate. Likewise, ordering of the children nodes of $E_\vee$ according to the equation

$$\frac{c(x_1)}{s(x_1)} \leq \ldots \leq \frac{c(x_n)}{s(x_n)} \tag{6.2}$$

minimizes the expected evaluation costs of $E_\vee$. However, this is true only under the independence assumption, and further, the work in [25] relies on the constant predicate costs assumption, too. Both the CNF and DNF based evaluation methods suffer from the overhead inflicted by the query normalization; the normalization yields queries that are exponentially blown up in their size.

In Muralikrishna's work [51], common subexpressions are identified by means of *merge graphs*. Given a Boolean expression in DNF, the disjunctive predicates are merged such that the resulting number of scans and joins is minimized. The work in [51] concerns itself with the identification of duplicate predicates in order to avoid a repeated evaluation of the same predicate. It is also shown that the problem of common subexpressions elimination for Boolean expressions is NP-complete. This work relies on both assumptions, IA and CC, and further, it requires the queries in DNF, thus it is associated with the overhead of query normalization.

The *bypass processing technique* was first introduced in [35] for evaluating Boolean expressions with expensive predicates in the context of object-oriented databases. The main idea behind this work centers around avoiding the evaluation of expensive predicates whenever this is possible. That is, whenever the outcome of the Boolean expression can be determined by evaluating other predicates that are less expensive, the expensive predicates are bypassed. The optimization algorithm in [35] is based on A^* *search*, allowing for global ordering of atomic predicates, thus comes close to the optimum. However, due to the exhaustive enumeration of predicate orderings, it is too expensive, therefore impractical for queries with more than few predicates. Further, their algorithm does not support CSE, and relies on IA and CC. The work in [12] extends the work in [35] by handling `NULL` values.

In the work by Kemper et al. [36], a heuristic based on Boolean Difference Calculus is used to optimize the evaluation of Boolean expressions. The heuristic orders selection predicates based on their influence on the query outcome and on their evaluation cost. The elimination of common subexpressions is not considered, and predicate selectivities are assumed to be independent. Moreover, this approach assumes constant predicate costs, too.

Chang et al. [8] take the DNF optimization scheme a step further. The union operators are pushed down and placed immediately after the join operators. This way, the expensive multi-way union operator (sitting on the top of the DNF plans) and merging the streams of tuples from each disjunct is not required. The multi-way union operator is the most expensive operator in DNF plans; it has to filter duplicate tuples coming from all the sub-streams corresponding to Boolean summands in a Boolean query. The approach in [8], however, works only in special cases, that is, when the selectivity of the join operator is very low (the join operator occurring immediately before the union operator). Besides requiring queries in DNF, this work also relies on both assumptions (IA, CC).

In the work by Chaudhuri et al. [10], two key ideas are presented: (1) complex predicate factorization and (2) predicate condition relaxation. Their main idea centers around exploiting the available indexes in order to create index-intersect and union plans. In addition to the requirement of available indexes, this work assumes both IA and CC.

An optimal algorithm for converting *decision tables* to *decision trees* was presented by Reinwald and Soland in [55]. Boolean expressions can be considered as a special case of this algorithm. Therefore, the algorithm in [55] can be used to find the optimal plan for evaluating a Boolean expression. The inner nodes of the resulting decision tree contain Boolean conditions, each having two branches (true/false), whereas the tree leaves contain the respective actions. The actions can be viewed as the outcome of the Boolean expression (i.e., true or false). This algorithm has a time complexity proportional to $O(n-k+1)^{2^{(k-1)}}$, where n denotes the number of candidate decision trees and k the number of partitioning steps. Such time complexity renders this algorithm impractical (for any practical application). Moreover, it assumes fixed costs for each condition and relies on the independence assumption, too.

An example decision tree is shown in Fig. 6.1. Each node is extended to another node (i.e., a Boolean condition), or an action. The processing of a decision table corresponds to the traversal of the decision tree, starting from the root node, and following the path all the way to a leaf node, based on the Boolean outcome of each intermediate node. The action corresponding to the leaf node is then carried out. Since there are many equivalent decision trees for a given decision table, the algorithm in [55] finds the one with the lowest costs.

The algorithm presented by Reinwald and Soland embraces the *Branch and Bound* strategy as in the *Traveling Salesman* algorithm [39]. The algorithm works by considering the set of all candidate decision trees that are equivalent to a given decision table. The set of candidate decision trees is partitioned into smaller subsets, and a lower bound on expected costs is assigned to each subset. Subsets are further partitioned into smaller subsets, and finally a subset

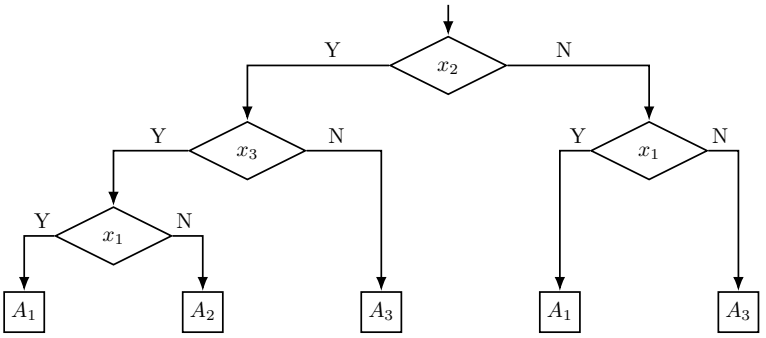


Figure 6.1.: A decision tree containing three Boolean conditions x_1, x_2, x_3

	Assumes		Supports		
	IA	CC	CSE	BMP	$\bar{\forall}$
Chang et al [8]	Yes	Yes	No	No	No
Chaudhuri et al. [10]	Yes	Yes	Yes	No	No
Claussen et al. [12]	Yes	Yes	No	No	Yes
Hanani [25]	Yes	Yes	No	No	No
Kemper et al. [35]*	Yes	Yes	No	No	Yes
Kemper et al. [36]	Yes	Yes	No	No	No
Muralikrishna [51]	Yes	Yes	Yes	No	No
Reinwald et al. [55]*	Yes	Yes	No	No	No

Table 6.1.: Overview of related work

containing only one element is found. This element corresponds to the optimal decision tree, with its expected costs lower or equal than the lower bound of all other subsets.

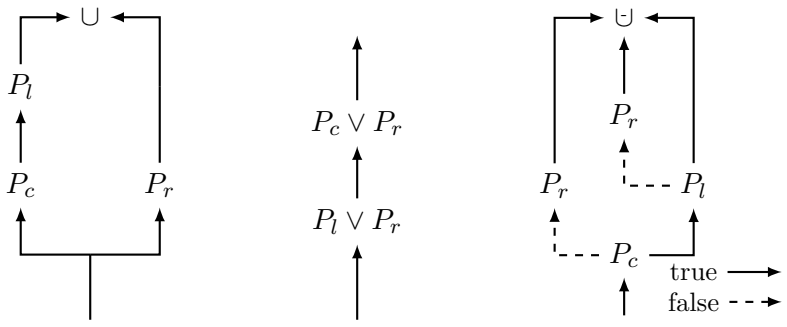
Table 6.1 summarizes the related work on the assumptions they make, the support of CSE, optimization of branch misprediction (BMP), and bypass processing ($\bar{\forall}$). Additionally, work that attains the optimum (in terms of plan quality) is marked with (*).

6.3. Preliminaries

In this section, we describe the preliminaries required to understand the optimization algorithms in this chapter.

6.3.1. Predicates

Predicates in our context are atomic, see Definition 5.1.1 in Chapter 5. Predicates can be connected by means of the conjunction AND ($\wedge$), e.g. $p_1 \wedge p_2 \wedge \ldots \wedge p_n$, or the disjunction OR ($\vee$), e.g. $p_1 \vee p_2 \vee \ldots \vee p_n$. Boolean expressions consisting of a mixture of both AND and OR connections can be transformed



(a) DNF plan (b) CNF plan (c) Bypass plan

Figure 6.2.: Evaluation plans for the query $(P_c \wedge P_l) \vee P_r$

to either normal form: conjunctive normal form (CNF) or disjunctive normal form (DNF).

A predicate query in CNF is composed of a conjunction of *Boolean factors*, where each Boolean factor contains predicates connected disjunctively:

$$\bigvee_{i=1}^{m} \bigwedge_{j=1}^{n} p_{i,j} := \underbrace{(p_{1,1} \vee \ldots \vee p_{1,n})}_{\text{1st Boolean factor}} \wedge \ldots \wedge \underbrace{(p_{m,1} \vee \ldots \vee p_{m,n})}_{m\text{th Boolean factor}}.$$

DNF is the dual of CNF. DNF is composed of a disjunction of *Boolean summands*, where each Boolean summand contains predicates connected conjunctively:

$$\bigwedge_{i=1}^{m} \bigvee_{j=1}^{n} p_{i,j} := \underbrace{(p_{1,1} \wedge \ldots \wedge p_{1,n})}_{\text{1st Boolean summand}} \vee \ldots \vee \underbrace{(p_{m,1} \wedge \ldots \wedge p_{m,n})}_{m\text{th Boolean summand}}.$$

6.4. Plan Construction Strategies

Bypass plans are best understood when compared against the traditional (non-bypass) plans. We present, therefore, three different plan construction strategies for disjunctive queries; DNF-based, CNF-based and bypass plans.

To illustratively compare these three different plan constructing strategies, we use a simple example query with three predicates over a fictional movies database:

```
(category = 'Action' AND length < 120) OR rating > 4
```

where the predicates are abbreviated as follows:

1. P_c: `category='action'`,
2. P_l: `length < 120`, and
3. P_r: `rating > 4`.

6.4.1. Traditional Plans: DNF and CNF

Traditionally, disjunctive queries in commercial databases are translated into either of DNF or CNF form. The DNF and CNF-based plans for our example query are shown in Figure 6.2a and 6.2b.

The construction of DNF-based plans constitutes to ordering of predicates within Boolean summands. The latter can be done efficiently as shown in [25]. However, this ordering relies on the independence assumption. Further, common subexpressions have to be identified in order to avoid redundant computations.

In DNF-based plans, there are as many tuple streams as there are summands in a DNF query. For our example query, two identical streams of tuples to be processed are created. One stream flows to the predicate P_c and the other to P_r. On top of DNF plans, there always exists a duplicate eliminating union operator ($\cup$), which merges and *eliminates* duplicate tuples. Due to identical streams of tuples which are produced in DNF plans, copies of tuples that pass the filtering conditions (predicates) from more than a single stream have to be eliminated. Of course, this approach with duplicate tuple elimination works only when the set semantics is desired. If otherwise the bag semantics (SQL semantics) is desired, tuples have to be appended with an identifier, e.g. RIDs, In the rest of the chapter, we will refer to the algorithm which produces DNF-based plans as DNFALG.

The CNF-based approach starts with a query in conjunctive normal form, consisting of a conjunction of Boolean factors (i.e., disjunctive terms). The optimization of CNF-based plans proceeds in two stages:

1. ordering of Boolean factors, and
2. ordering of predicates within Boolean factors.

Hanani in [25] shows how to determine an optimal ordering of (1) and (2). The ordering according to Hanani's [25] equations (cf. Eq. 6.1, 6.2, in related work section) yields the optimal plan only under the independence assumption.

In contrast to DNF-based plans, CNF-based plans do not need the union operator. CNF-based plans have the disadvantage of repetition of selection predicates in all the stages of the query plan. As it can be seen in the plan depicted in Figure 6.2b for our example query, the predicate P_r occurs in both stages, this way adding unnecessary costs. We shall call this algorithm CNFALG.

6.4.2. Bypass Plans

Each bypass selection maintains two *disjoint* output streams of tuples, namely the *true* and the *false* stream. The idea is to discard those tuples that do not make it to the result set as early as possible. The bypass technique is not new, it was already used in [12, 35].

Consider again our example query, together with its depicted bypass plan in Figure 6.2c. Tuples that satisfy the predicate P_c are pipelined to the next predicate, namely P_l, and if they satisfy this predicate, too, they are part of

the final result. On the other hand, tuples that do not satisfy P_c bypass P_l altogether, as the evaluation of P_l is unnecessary, therefore, they are pipelined directly to predicate P_r instead. These tuples correspond to the false stream of P_c. Bypass processing is instrumental in avoiding expensive predicates (e.g., predicates with the SQL LIKE, UDF calls, etc) whenever their evaluation is not required. Furthermore, in contrast to DNF-based plans, the union operator ($\uplus$) sitting on top of the bypass plan does not have to perform the extra work of duplicate elimination, as tuple streams in bypass plans are disjoint. That is, the union operator in bypass plans is a simple multi-way merge operator, which is much cheaper to evaluate than the union operator with the duplicate elimination ($\cup$). To that end, bypass plans offer a vast optimization potential which has been experimentally shown in [12, 35], and reconfirmed by our experiments (see Section 6.6).

6.5. A Heuristic Optimization Algorithm for Boolean Expressions

In this section, we present a heuristic optimization algorithm for disjunctive predicates. We initially provide an overview of how the algorithm works and then we give a detailed description of the algorithm.

6.5.1. Overview of the Algorithm

The main idea behind the algorithm which we are going to present in this section centers around (1) ordering (optimally) the predicates within each summand, and (2) ordering of the summands themselves, in addition to constructing valid bypass plans. For illustration purposes, let us consider the following example query:

$$(x_1 \wedge x_2 \wedge x_3) \vee (x_2 \wedge x_5 \wedge x_6).$$

Illustration of the bypass plan construction for this example query is shown in Figure 6.3. The algorithm starts by building a plan consisting of only a scan operator (depicted in Figure 6.3a), as all other operators are built on top of it. Next, each Boolean summand given in the query is picked as a candidate for the next partial plan that will be built on top of the scan operator. Since a Boolean summand contains a conjunction of predicates, the (partial) plan evaluating these predicates is built by means of the DPSel algorithm (cf. Figure 6.6). Note that the DPSel algorithm shown in Figure 6.6 is a modified version of the original algorithm shown in Chapter 5, adapted for bypass plan generation.

For our example query, DPSel returns a partial plan on top of the scan operator, as illustrated in Figure 6.3b. In the newly created plan, there is a *negative edge* (denoted by the minus '$-$' sign) for each selection operator. The negative edges stand for the false stream of tuples, and we call them *false branches*. In each one of these false branches, the partial plan evaluating the next summand is built in the same fashion as on top of the scan operator (cf. Figure 6.3d, 6.3e). In the process, however, we need to be careful with details,

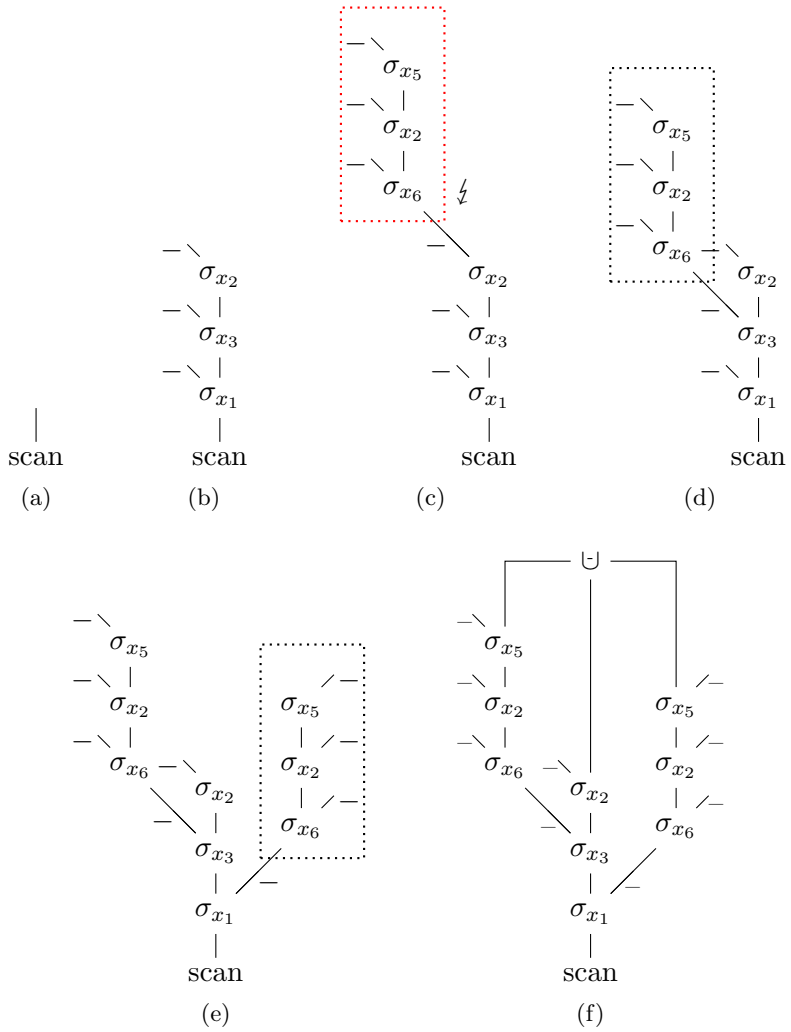


Figure 6.3.: Bypass plan construction for the example query $(x_1 \wedge x_2 \wedge x_3) \vee (x_2 \wedge x_5 \wedge x_6)$

```
BypassPlanGen(E)
    // Input: a set of boolean summands E = {S_1, ..., S_n}
    // Output: best plan
1   bestplan.branches = {scan(R)}
2   return Optimize(bestplan, E)
```

Figure 6.4.: Pseudocode for BypassPlanGen

```
Optimize(e, E)
    // Input: plan e
              a set of boolean summands E = {S_1, ..., S_n}
    // Output: best plan
 1  bestplan = NULL
 2  for each S_i ∈ E
 3      B = ∅
 4      for each b ∈ e.branches
 5          F = {false preds. in b}
 6          T = {true preds. in b}
 7          if F ∩ S_i == ∅ ∧ T ⊉ S_i
 8              B = B ∪ DPSel(b, S_i \ T).branches
 9      e_t = an empty plan
10      e_t.branches = B
11      e_t = Optimize(e_t, E \ {S_i})
12      if bestplan == NULL ∨ C(bestplan) > C(e_t)
13          bestplan = e_t
14  return bestplan
```

Figure 6.5.: Pseudocode for Optimize

such as duplicate predicates occurring among different summands, as well as common subexpression elimination.

6.5.2. The Algorithm in Detail

In this section, we present the details of the inner workings of our optimization algorithm coined BypassPlanGen, abbreviated – BypassPlanGen. Its pseudocode is shown in Figure 6.4.

BypassPlanGen accepts as an input a query in disjunctive normal form, and further, selectivities must be provided for all subsets of predicates occurring in the query. Selectivity estimates for subsets of predicates can be computed by means of entropy maximization [42], graphical models [63] or by means of the sampling method shown in Chapter 8.

BypassPlanGen starts (line 1) by creating a plan – *bestplan*, and the branch of a plan containing only the scan operator is stored in its *branches* container.

DPSel($branch, S$)

```
    // Input: a branch
              a set S = {p_0, ..., p_{n-1}} of predicates
    // Output: optimal subplan on top of branch
 1  DP = an empty DP table, size → 2^n
 2  DP[∅] = branch
 3  for 0 ≤ i < 2^n − 1 ascending
 4        S' = {p_k ∈ S | (⌊i/2^k⌋ mod 2) = 1}
 5        for each p_j ∈ S \ S'
 6              e = an empty plan
 7              if S' == ∅ ∧ DP[S'] ≠ scan(R)
 8                    e = BuildPlans(p_j, DP[S'], FALSE)
 9              else
10                    e = BuildPlans(p_j, DP[S'], TRUE)
11              StoreSolution(e, S' ∪ {p_j}, DP)
12  return DP[S]
```

Figure 6.6.: Pseudocode for DPSel

BuildPlans($p, e, stream$)

```
    // Input: a selection predicate p
              an expression (partial plan) e
              boolean flag stream
    // Output: (partial) plan
1   X_e = ∪_{p_j ∈ e} X_{p_j}
2   X_{p|e} = X_p \ X_e // outstanding maps
3   if e == scan(R)
4         return σ_{p_i}(X_{p|e}(e))
5   elseif stream == TRUE
6         e = σ_{p_i}(X_{p|e}(σ^+_{p_j}(e')))
7   else
8         e = σ_{p_i}(X_{p|e}(σ^−_{p_j}(e)))
9   return e
```

Figure 6.7.: Pseudocode for BuildPlans

StoreSolution(e, S, DP)

```
1  if DP[S] == NULL ∨ C(DP[S]) > C(e)
2        DP[S] = e
```

Figure 6.8.: Pseudocode for StoreSolution

A plan is essentially a container of branches. In line 2, the procedure OPTIMIZE is called with arguments *bestplan*, and the set of Boolean summands E.

The procedure OPTIMIZE (cf. Figure 6.5) works recursively by trying all the permutations of Boolean summands in E. In line 1, an empty plan is created, where the cheapest plan found will be stored and returned to the caller. Line 2 iterates over all the summands in E, and in line 3 an empty container of branches is created. In line 4, the algorithm iterates over the branches of the input plan e. We need to consider the possibility of building a partial plan evaluating the current summand S_i on top of each branch in e. However, before we do that, we need to check for *common predicates*. That is, we need to make sure that S_i does not contain a predicate which is evaluated by a selection operator with a non-empty false branch occurring below b, or by the selection operator that b belongs to. By a *non-empty* branch we mean a branch that is connected to some other operator, e.g., the false branch of the operator σ_{x3} in Figure 6.3d is not empty, as it is connected to σ_{x6}, whereas the false branch of the operator σ_{x2} (in Figure 6.3d) is empty. In the same fashion, we need to make sure that the predicates in the summand S_i have not been already evaluated by the selection operator in b or selection operators situated below b. To better illustrate this, consider the example plan shown in Figure 6.3c. There is no point in building the partial plan (shown in the red-dotted rectangle in Figure 6.3c) evaluating predicates in $S_i = \{x_2, x_5, x_6\}$ on top of the false branch of the bypass selection operator σ_{x_2}, as the result will be false anyway.

Line 7 of procedure OPTIMIZE makes sure that no such unnecessary branches are added to the plan. In the set F (line 5), we store the predicates evaluated by the selection operator that b belongs to, and all the predicates belonging to the selection operators occurring below b, that have a non-empty *false* branch. In the set T (line 6) on the other hand, we store all the predicates belonging to the selection operator in b and selection operators situated below b, i.e., *true* branches.

Common predicate detection is crucial for building efficient plans: by detecting such predicates, we can omit entire (unnecessary) bypass branches, and this way drastically reduce the plan costs, and at the same time reduce the search space. For the example partial plan in Figure 6.3b, and $S_i = \{x_2, x_5, x_6\}$, $b = \sigma_{x_2}$, we have $F = \{x_2\}$, $T = \{x_1, x_2, x_3\}$, thus the condition in line 7 results with a false outcome ($F \cap T \neq 0$). That is, there will be no plan built on top of the false branch of σ_{x_2}.

If there are no common predicates between the set F and T, DPSEL algorithm is called with the branch b, and the current summand S_i (minus the predicates in T) as arguments (line 8). DPSEL in turn will find an optimal (partial) plan for evaluating the conjunction of predicates in S_i on top of the branch b. To that end, the false branches of the newly created partial plan by DPSEL are returned and added into the container – B. Going back to our example query, if DPSEL was called with arguments $b = scan(R)$, and $S_i = \{x_2, x_5, x_6\}$, a partial plan is built on top of the scan operator, containing three false branches, depicted in Figure 6.3b. These false branches in turn, will be stored in the container B such that on the next recursive invocation of OPTIMIZE, we can build the next Boolean summand on top of each branch stored in B, given that the condition

in line 7 is fulfilled. This has been illustrated in Figure 6.3d–6.3e, where the newly added partial plans have been depicted in dotted rectangles. Details of DPSel are given in Section 6.5.3.

In line 9, another empty plan (e_t) is created, which will hold the plan evaluating the current Boolean summand S_i, and in line 10, e_t gets the branches stored in B.

In line 11, the procedure Optimize calls itself with the new plan e_t and the remaining summands. Line 12 compares the cost of the plan currently found against *bestplan*. The dominating plan (i.e., the cheapest plan) is kept in *bestplan*, and all other plans are pruned. When the recursion ends, the *bestplan* is returned to the caller.

6.5.3. Optimization of Conjunctive Predicates in Boolean Summands

The partial plans for conjunctive predicates in Boolean summands are built by means of the DPSel algorithm (cf. Figure 6.6). This algorithm produces an optimal plan for conjunctive predicates, generating solutions in a bottom-up fashion, using dynamic programming. Since Boolean summands are composed of conjunction of predicates, we can use the DPSel to optimize these predicates. To this end, we have extended DPSel for the purpose of generating bypass plans, as the original version of DPSel (shown in Chapter 5) supports neither queries with disjunctions nor bypass plan creation.

DPSel starts by initializing an empty DP table and storing the input branch in it (cf. lines 1-2 in Figure 6.6). Operators evaluating selection predicates in the input Boolean summand S are built on top of this branch. The loop in line 3 iterates over all subsets S' of predicates in the summand S. The loop in line 5 iterates over the predicates in S which are not in S'. These are the new predicates that are not yet included in the existing partial plans stored in the DP table.

Adding the new predicates to the existing (partial) plans is the responsibility of the BuildPlans procedure shown in Figure 6.7. The BuildPlans procedure takes as an input a predicate p, an existing partial plan e, and a Boolean flag *branch*, indicating whether the new predicate is to be added to the regular (true) branch or the false branch of the top bypass selection operator in the existing input partial plan (e). In multiple predicate Boolean expressions, one should take care of redundancies in terms of column accesses, user-defined function calls, etc. For example, in the query `EMP.age > 20 AND EMP.age < 51`, the attribute values of `EMP.age` are needed in two places, hence the common subexpression. We can, however, use a single map operator (χ) accessing the values of `EMP.age` (cf. Figure 6.9), and this way eliminating the redundancy. Recall that the map operator is used for dereferencing (i.e., accessing) column values (cf. Chapter 3).

We differentiate the common subexpression elimination (CSE) from the common predicates elimination in that the former allows for column access optimization, whereas the latter allows for false branch elimination when different summands have predicates in common. Both are of a fundamental importance

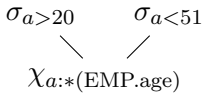


Figure 6.9.: Dependency graph for the example query

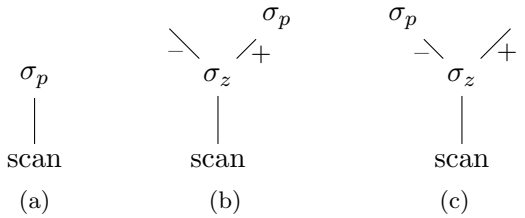


Figure 6.10.: Illustration of building bypass plans

to building efficient plans.

In the procedure BuildPlans, the set of map dependencies on which each input predicate p depends, as well as CSE, are taken care of in lines 1,2. For the sequence of selections in the partial plan e, their already executed map dependencies are denoted by $X_e = \cup_{p_i \in e} X_{p_i}$, whereas the map dependencies of the input predicate p, which are still to be executed, are denoted by $X_{p|e} = X_p \setminus X_e$.

To illustrate CSE implementation in our algorithm, consider again our example query shown in Figure 6.3, and let the partial plan $e = \sigma_{x_2}(\sigma_{x_3}(\sigma_{x_1}(scan)))$, and $p_i = x_2$. For the partial plan e, we have $X_e = \{x_1, x_2, x_3\}$; these are the already executed map operators (cf. Section 3.1.1) for predicates x_1, x_2, x_3. Now, we need to evaluate the new predicate p_i on top of the existing plan e, but before we do that, we need to make sure that its map dependencies are fulfilled. Therefore, for $p_i = x_2$, we have:

$$X_{p|e} = X_p \setminus X_e = \{x_2\} \setminus \{x_1, x_2, x_3\} = \varnothing$$

which results in an empty set, this way *ensuring* that we do not add a redundant map operator to our plan and unnecessarily increase the plan costs.

After the map operators and CSE are taken care of, three cases are distinguished, covered in lines 3-8 of the BuildPlans procedure (cf. Figure 6.7):

1. If the input partial plan (e) is a scan operator (cf. line 3), then the only way to evaluate the input predicate p is by a selection operator which is added on top of the scan operator. This has been illustrated in Figure 6.10a.

2. If the branch flag is set to true (cf. line 5), it follows that the input partial plan (e) must contain a bypass selection operator as the top operator (i.e., not a scan operator), then the predicate p is evaluated by a new bypass selection operator added on top of the true branch of the selection operator in e, as depicted in Figure 6.10b.

3. The last case implies (cf. line 7) that the input branch flag is set to false, therefore, the selection operator evaluating p is added on top of the false branch of the top operator in e, as depicted in Figure 6.10c.

Finally, the newly constructed plan is returned to the caller.

In line 6 of DPSel, an empty plan is created with the purpose of holding the plan which will be subsequently returned by the BuildPlans procedure. Line 7 checks if the set S' is empty and at the same time makes sure that the input branch is not a scan operator. If this condition if satisfied, BuildPlans procedure is called with the false flag, indicating that it should build the false branch of the bypass selection operator in $DP[S']$. The reason for the check in line 6 is to make sure that the procedure BuildPlans is not requested to build a false branch on top of a scan operators as obviously this operator does not have a false branch. If condition in line 7 is not satisfied, it entails that the top operator of the input branch is a scan operator, therefore the algorthim should build on top of its false branch. Recall that in the DP table for the key $S' = \varnothing$ we have associated the input *branch* (cf. line 2).

Line 11 of DPSel algorithm passes the plan in e to the auxiliary procedure StoreSolution (see Figure 6.8), which in turn stores the *dominating* plan (the plan with the lowest cost) in the DP table. Plans produced by our algorithm are costed according to the cost model shown in Chapter 4. Our cost model takes into account the branch misprediction. The branch misprediction in turn depends on predicate selectivities. As stated already in Section 6.5.2, predicate selectivities for all the subsets of predicates are given as an input to the algorithm.

By the time the main loop (lines 3-11) of DPSel exits, the best plan with the optimal cost for evaluating the selection predicates in S on top of the input *branch* is found and stored in the DP table. In line 12, the best plan is fetched from the DP table and is returned to caller, i.e., to the procedure Optimize.

The time complexity of BypassPlanGen is $O(k!\, m^{k-1})$, whereas its space complexity is $O(m^k)$. Note that k stands for the number of Boolean summands in a query, and m for the number of predicates in the largest (single) Boolean summand. Although the time complexity of BypassPlanGen seems rather high, it is still asymptotically much lower than the superexponential algorithm given by Reinwald and Soland (cf. Section 6.2), or an algorithm that enumerates all permutations of predicates – $O(n!)$ for n predicates. Further, the gains in plan quality when choosing BypassPlanGen over normal-form based algorithms outweigh by far its costs as we will show in the experimental evaluation of the algorithm in Section 6.6.

6.6. Evaluation of the Heuristic Algorithm

The evaluation of predicates in data warehouses has become the major bottleneck for decision support queries [31]. We show in this section that there is a huge optimization potential not harvested by optimization algorithms typically used in RDBMSs. For the experimental evaluation of our optimization algorithm – BypassPlanGen, we have compared it against two widely used

algorithms DNFALG and CNFALG (cf. Section 6.4). Since DPSEL [33] produces an optimal plan for conjunctive predicates, we applied it to the original DNFALG algorithm, such that for each tuple stream an optimal local plan is produced. Recall that in DNF plans (cf. Section 6.4) there are as many tuple streams as there are Boolean summands in a query. We termed this new algorithm DNFDP. DNFDP, thanks to DPSEL minimizes the branch misprediction penalty, and in addition, it applies CSE in each Boolean summand (locally). We introduced DNFDP with the goal of improving the chances of a DNF-based algorithm against the algorithm presented in this chapter – BYPASSPLANGEN.

For testing the algorithms, we have performed three different sets of experiments over three different data sets. We conclude the Experiments section with a comparison of the running times of the four algorithms. All experiments were run single-threaded on a machine with an Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz processor. The machine was equipped with 264 GB of main memory, and ran a 64 bit Arch Linux operating system. All the algorithms were implemented in `C++`, and compiled using `g++` (version 6.2.1).

6.6.1. Forest Dataset

We have already introduced the Forest [14] dataset in the Evaluation section of Chapter 5. Recall that the Forest [14] dataset contains data about US forests, materialized in a relation with 54 attributes, and 581.012 tuples. This rather wide relation validates the importance of optimizing Boolean expressions. We have used three different queries, each containing 6 predicates in total and varying number of Boolean summands (for queries in DNF) or Boolean factors (for queries in CNF), respectively.

The predicates were simple range predicates of the form $c_1 \leq attr_i \leq c_2$, were c_1, c_2 denote integer constants. As we do not rely on the IA, selectivity estimates for all the subsets of predicates where derived by means of the very efficient sampling method shown in Chapter 8.1.

We have compared all the four algorithms by generating randomly 1000 queries over randomly chosen attributes of the Forest relation. For all the experiments, we have transformed the queries into both normal forms: DNF and CNF. This way, we did not give any unfair advantage to any algorithm over the rest.

Two different sets of experiments were performed. The first set contained only inexpensive predicates, i.e., the cost of the map operators that predicates depended on was set to 1. That is, all predicates had *equal* costs. For the second set, we assigned random cost values to the map operator, uniformly distributed in the range [1, 100]. That is, predicates had *varying* costs. For each query, there were 100 different such random cost assignments to the map operators that each predicate depended on.

Queries with equal predicate costs (no common predicates)

The results of this experiment are shown in Table 6.2 and Table 6.3. Note that 'B.s.' and 'B.f.' denote the number of Boolean summands (for queries in

Query type	DNF					
Costs	Equal			Varying		
# B. s.	1	2	3	1	2	3
DNFalg	1.6	3	4.3	22.2	9.4	29.1
DNFdp	1.0	2.3	3.6	1.0	9.3	29.1
CNFalg	5.4	5.6	14.3	79.2	243.2	585.8

Table 6.2.: Relative optimization potential (in factors!) of BypassPlanGen vs. DNFalg, DNFdp and CNFalg over the Forest dataset

Query type	CNF					
Costs	Equal			Varying		
# B. s.	1	2	3	1	2	3
DNFalg	6.9	15.1	17.1	89.7	186.8	290.1
DNFdp	6.9	12.5	14.7	89.7	185.8	288.9
CNFalg	1.0	4	4.1	2.09	38.5	49.2

Table 6.3.: Relative optimization potential (in factors!) of BypassPlanGen vs. DNFalg, DNFdp and CNFalg over the Forest dataset

Query type	DNF					
Costs	Equal			Varying		
# B. s.	1	2	3	1	2	3
BypassPlanGen	5.4	9	7.8	65.2	139.7	123.7
DNFalg	6.1	13.1	16.7	75.5	174.3	232.8
DNFdp	5.4	11.1	14.6	62.2	168.7	231.6
CNFalg	10.1	25.9	24.6	137.2	615.4	781.5

Table 6.4.: Average time-per-tuple (ns) for query plans over the Forest dataset

Query type	CNF					
Costs	Equal			Varying		
# B. s.	1	2	3	1	2	3
BypassPlanGen	4	4.7	7.5	27.8	32.4	42.5
DNFalg	18.7	44.3	47	321.5	723.6	746.5
DNFdp	18.7	38.2	38.2	321.5	713.9	720.8
CNFalg	4	6.6	10.3	28.2	86.68	160.1

Table 6.5.: Average time-per-tuple (ns) for query plans over the Forest dataset

DNF), and number of Boolean factors (for queries in CNF), respectively. For the definition of Boolean summands, respectively Boolean factors please refer to Section 6.3. As the plan costs varied greatly, the difference on plan qualities between the algorithms is shown in *factors* relative to BypassPlanGen. That is, over all runs, we recorded the *maximum* deviation factor on the plan quality of the plans produced by normal form algorithms (DNFalg, DNFdp, CNFalg) relative to the plans produced by BypassPlanGen algorithm.

Starting with queries in DNF with inexpensive predicates (cf. Column 'Equal' in Table 6.2), and queries containing a single Boolean summand, BypassPlanGen beats DNFalg by a factor of over 1.6 and CNFalg by a large factor of 5.4. Since the query had only a single summand, and since both BypassPlanGen and DNFdp employ the DPSel algorithm, they both produced the optimal plan, hence equal costs. That is, queries containing a single Boolean summand contain predicates connected conjunctively (no OR operations), therefore, there is no opportunity for bypassing selection operators in the resulting plans. Continuing with queries with 2 and 3 Boolean summands, BypassPlanGen beats the other algorithms by large factors, namely DNFalg by a factor 3 and 4.3, DNFdp by a factor of 2.3 and 3.6, and CNFalg by 5.6 and 14.3, respectively. These are very large factors, considering that the predicates were cheap to evaluate, and there were only 6 predicates in total. In case of the CNFalg, the factors of difference in plan qualities relative to BypassPlanGen are much larger when the query is given in DNF, as it has first to be transformed into CNF, this way inducing multiple duplicate predicates across Boolean factors. The same disadvantage applies to DNF-based algorithms when queries are given in CNF (cf. Table 6.3). Note that when converting a query from one normal form to another, the query blows-up exponentially in its size as a result. Even for queries in CNF, the factors are very large relative to BypassPlanGen, namely 6.9 for DNFalg, and DNFdp, for a single Boolean factor, all the way to staggering factors of over 14 for three Boolean factors (for cheap predicates)! For a single Boolean factor, both CNFalg and BypassPlanGen produced the same results, as the query contained only OR operations. With the increase of the number of Boolean factors, however, BypassPlanGen starts producing plans that are cheaper than plans produced by CNFalg by a factor of 4 for 2 and 3 Boolean factors (for cheap predicates). To this end, regardless of the normal form of the query (i.e., DNF or CNF), BypassPlanGen is the clear winner by large factors.

Varying predicate costs (no common predicates)

For predicates with *varying* costs, the difference on plan qualities between BypassPlanGen and normal form based algorithms is much greater (cf. columns 'Varying' in Table 6.2 and Table 6.3). Starting with a single Boolean summand, BypassPlanGen beats DNFalg by a large factor of 22, and CNFalg by a huge factor of 79. Same as with cheap predicates, DNFdp produces the same results as BypassPlanGen due to the fact that there was only a single summand, therefore no OR operators. Continuing with 2 and 3 Boolean summands, BypassPlanGen is again the clear winner by very large margins. It

beats both DNF algorithms by factors of over 9 and 29, and CNFALG by factors of 243 and 585, respectively. For queries in CNF form (cf. columns 'Varying' in Table 6.3), the situation is not better for the heuristics. Starting with a single Boolean factor, BYPASSPLANGEN produced plans that are cheaper than plans produced by DNFALG and DNFDP by a factor of 89. CNFALG on the other hand lost by a factor of 2 against BYPASSPLANGEN. For queries with 2 and 3 Boolean factors, BYPASSPLANGEN beats DNF-based algorithms by factors of over 185 and 288! It beats CNFALG too, by factors of 38 and 49, respectively.

To that end, the gaps on plan qualities are very large, despite the fact that the total number of predicates was limited to only 6 predicates. We initially thought that because we have picked the maximum optimization potential over all the queries, it might be the case that BYPASSPLANGEN is performing so well only for some few particular queries, and for the rest of queries the difference might not be so large. We repeated the same experiments, but this time we took the average time-per-tuple, over all queries. The results of this experiment are shown in Table 6.4 for queries in DNF, and Table 6.5 for queries in CNF, respectively. BYPASSPLANGEN nevertheless shows much better query times. For two Boolean summands (query in DNF), the plan quality difference between BYPASSPLANGEN vs. DNFALG and DNFDP is 31%, and 18% respectively. For three summands, plans produced by BYPASSPLANGEN are 53% cheaper than those produced by DNFALG, and 46% cheaper than those produced by DNFDP! It is interesting to note that for queries with 2 Boolean summands, the plans produced by BYPASSPLANGEN are more expensive than for queries with 3 Boolean summands. The reason for such behavior is that for 2 Boolean summands, BYPASSPLANGEN has less opportunity to bypass predicates than when there are 3 Boolean summands in a query. It follows that the gap on plan qualities—between BYPASSPLANGEN and heuristics—would increase even further if we would increase the number of summands, respectively the number of predicates. For queries in CNF with 2 and 3 Boolean factors, BYPASSPLANGEN produced plans that are by 27% cheaper than those produced by CNFALG.

For queries with varying predicate costs, regardless of the query form (DNF, CNF) or the number of Boolean summands/factors, the gap on plan qualities remains large in favor of BYPASSPLANGEN (cf. column 'varying' in Table 6.4 and Table 6.5).

Queries with Common predicates

We have repeated the same experiment over the same data set, with the difference that in this experiment we introduced duplicate predicates in our queries. That is, we generated a pool with 6 predicates in total, and created $100k$ different queries with random constants c_1, c_2 (recall that predicates over the Forest data set were of the form $c_1 \leq attr_i \leq c_2$). Each query in turn contained 2 and 3 Boolean summands/factors. Predicates for each query were picked randomly from the pool of predicates, this way giving a leeway to duplicate predicates. We have omitted the results for queries with a single Boolean summand/factor, as such queries are pure conjunctive/disjunctive. The results of this experiment are shown in Table 6.6.

Query type	DNF				CNF			
Costs	Equal		Varying		Equal		Varying	
# B. s./B. f.	2	3	2	3	2	3	2	3
DNFALG	3.2	6.8	46	81	19	21	244	331
DNFDP	3.2	5.7	46	81	15	16	243	330
CNFALG	14	97	1021	2574	4.3	6	60	100

Table 6.6.: Relative optimization potential (in factors!) of BYPASSPLANGEN vs. DNFALG, DNFDP and CNFALG over the Forest dataset (common predicates)

Query type	DNF				CNF			
Costs	Equal		Varying		Equal		Varying	
# B. s./B. f.	2	3	2	3	2	3	2	3
BYPASSPLANGEN	7.2	5	103	67	4.1	4.4	36	47
DNFALG	12	15	170	225	29	34	489	517
DNFDP	10	13	165	222	26	29	483	505
CNFALG	18	21	391	652	7	10	98	155

Table 6.7.: Average time-per-tuple (ns) for query plans over the Forest dataset (common predicates)

For 2 and 3 Boolean summands and predicates with *equal* costs, BypassPlanGen beats DNFalg by a factor of 3.2 and 6.8 respectively. DNFdp produced better plans than DNFalg due to DPSel which it employs, however, it is still inferior to BypassPlanGen by factors of 3.2 and 5.7. As expected, CNFalg produced very poor plans (dominated by plans of BypassPlanGen by factors of 14 and 97) as the queries were in DNF. For predicates with *varying* costs, BypassPlanGen beats DNF-based algorithms by very large factors, namely 46 and 81! As expected, CNFalg again produced extremely poor plans.

For queries in CNF, situation is not much better, except that the tables have turned in favor for CNF vs. DNF-based algorithms. Nevertheless, BypassPlanGen is the clear winner by large factors even for queries in CNF, and for both cases, predicates with equal and varying costs (cf. columns under the label 'CNF' in Table 6.6). Such large differences on plan qualities between BypassPlanGen and normal form based algorithms are due to the ability of BypassPlanGen to completely eliminate plan branches, whenever there are shared predicates among different summands (cf. Section 6.5.2). This in addition to the advantages of bypass plan generation.

The average time-per-tuple for queries with duplicate predicates are shown in Table 6.7. For queries in DNF with equal costs, with 2 and 3 Boolean summands, BypassPlanGen produces plans that are cheaper by 41% and 66% than the plans produced by DNFalg, and by 32%, 62% than the plan produced by DNFdp, respectively. As the queries were in DNF form, CNFalg produced far inferior plans. For queries in DNF with varying costs, BypassPlanGen beats DNFalg by 39% and 70%, and DNFdp by 37% and 69%.

For queries in CNF with equal costs, containing 2 and 3 Boolean factors, BypassPlanGen produced plans that are cheaper by 41% and 56% than plans produced by CNFalg. Plans for DNF based algorithms were by far inferior to BypassPlanGen for queries in CNF, as expected. For varying costs, BypassPlanGen produced plans that are 62% and 69% cheaper than those produced by CNFalg.

6.6.2. Predicates with Random Selectivities

For this experiment, selectivities for single predicates P_i and pairs $(P_i \wedge P_j) \forall i, j$ were generated randomly, uniformly distributed in the range $[0, 1]$. Their consistency was ensured by means of PDHGMp [47]. When randomly generating predicate selectivities, there can be inconsistencies, therefore we have used PDHGMp algorithm for generating consistent predicate selectivities. For the rest of predicates $\wedge_{i \in I} P_i, I \subseteq \{1, \ldots, n\}$, their joint selectivities were generated by the principle of *maximum entropy* (ME) [42]. We have again used queries with a maximum of 6 predicates, and as in previous experiments, we did two sets of experiments: one experiment included predicates with *equal* costs, whereas the other one predicates with *varying* costs. A total of $1k$ queries were used, in both normal forms (DNF, CNF), and each with 2 and 3 Boolean summands/factors. The results of this experiment are shown in Table 6.8. The plan difference of normal form based algorithms is shown in factors relative to BypassPlanGen.

Query type	DNF				CNF			
Costs	Equal		Varying		Equal		Varying	
# B. s./B. f.	2	3	2	3	2	3	2	3
DNFALG	2.5	3.7	2.7	5.4	16	20	91	122
DNFDP	2	3.1	2.7	5.4	14	15	89	119
CNFALG	22	260	94	866	2.8	3.6	11	20

Table 6.8.: Relative optimization potential of BYPASSPLANGEN vs. DNFALG, DNFDP and CNFALG, joint predicate selectivities generated by ME principle

Query type	DNF				CNF			
Costs	Equal		Varying		Equal		Varying	
# B. s./B. f.	2	3	2	3	2	3	2	3
BYPASSPLANGEN	11	10	158	127	6.3	6.6	38	38
DNFALG	23	25	242	274	73	86	815	941
DNFDP	19	21	228	264	62	70	789	892
CNFALG	104	715	3758	29895	11	15	132	190

Table 6.9.: Average time-per-tuple (ns), joint predicate selectivities generated by ME principle

As it can be seen in Table 6.8, for queries in DNF with equal predicate costs, BYPASSPLANGEN produces cheaper plans than DNF-based algorithms by factors of over 2 and 3, respectively. For predicates with varying costs, factors have increased to 2.7 and 5.4. As expected, CNFALG produced far inferior plans due to the queries being in DNF.

For queries in CNF, BYPASSPLANGEN proved again to be the best alternative. It beats CNFALG by factors of over 2 and 3 for cheap predicates (equal costs), and by factors of over 11 and 20 for predicates with varying costs. DNF-based algorithms were vastly inferior due to the queries being in CNF.

The average plan costs (in time-per-tuple) have been shown in Table 6.9; they confirm that BYPASSPLANGEN produces cheaper plans, regardless of the number of Boolean summands/factors or predicate costs. The gaps between the DNFALG and CNFALG relative to BYPASSPLANGEN in this experiment have even further increased compared to the experiment with the Forest dataset.

Queries with common predicates

In the following, we present the results of the algorithms optimizing queries which contain shared predicates, for the dataset generated by the principle of ME. As with the Forest dataset experiment, we generated a pool with 6 predicates in total, and 100k random queries, with 2 and 3 Boolean summands/factors. Predicates were picked randomly from the pool of predicates. The

Query type	DNF				CNF			
Costs	Equal		Varying		Equal		Varying	
# B. s./B. f.	2	3	2	3	2	3	2	3
DNFalg	3.5	6.3	31	37	16	20	113	140
DNFdp	2.9	5.3	29	36	14	15	111	132
CNFalg	24	303	136	2233	2.9	5.1	13	25

Table 6.10.: Relative optimization potential of BypassPlanGen vs. DNFalg, DNFdp and CNFalg, joint predicate selectivities generated by ME principle (common predicates)

Query type	DNF				CNF			
Costs	Equal		Varying		Equal		Varying	
# B. s./B. f.	2	3	2	3	2	3	2	3
BypassPlanGen	10	7.7	109	71	6.3	6	49	54
DNFalg	20	24	212	260	48	59	504	652
DNFdp	17	21	203	253	42	49	490	623
CNFalg	41	219	985	7218	10	14	128	188

Table 6.11.: Average time-per-tuple (ns), joint predicate selectivities generated by ME principle (common predicates)

results of these experiments are shown in Table 6.10.

For queries with two and three summands, and for predicates with equal costs, BypassPlanGen beats DNF-based algorithms by factors of over 2 and 5, respectively. These are large gaps given that the predicates were cheap to evaluate. For varying costs predicates, the gaps have increased to whopping factors of 29 and 36! For queries in CNF, BypassPlanGen is again the clear winner. It beats CNFalg by factors of over 2 and 5 for cheap predicates, and by factors of over 13 and 25 for predicates with varying costs.

The average plan costs over all queries used in this experiment are shown in Table 6.11. Comparing these figures with the ones in Table 6.9 (no common predicates), we can see that BypassPlanGen has produced cheaper plans, whereas the plan costs for the normal form based algorithms have actually increased. The reason why the plan costs have increased for the normal form based algorithm is that in this experiment we used more queries – 100k instead of 1000. Whereas the reason why BypassPlanGen produced cheaper plans comes as a result of the optimization of common predicates.

We conclude that the detection and elimination of duplicate predicates plays a crucial role in optimization of Boolean expressions, and thus it cannot be ignored.

6.6.3. CH-benchmark

In this section, we present the results of comparing the algorithms by using the CH-benchmark [13, 62] workload. CH-benchmark is a complex, mixed workload benchmark, devised with the goal of closing the gap between TPC-C (for OLTP) and TPC-H (for OLAP).

For the purpose of testing our algorithms, we have picked Query 19 of this benchmark:

```
SELECT  SUM(ol_amount) AS revenue
FROM  orderline, item
WHERE (
  ol_i_id = i_id
  AND i_data LIKE '%a'
  AND ol_quantity >= 1
  AND ol_quantity <= 10
  AND i_price BETWEEN 1 AND 400000
  AND ol_w_id IN (1,2,3)
) OR (
  ol_i_id = i_id
  AND i_data LIKE '%b'
  AND ol_quantity >= 1
  AND ol_quantity <= 10
  AND i_price BETWEEN 1 AND 400000
  AND ol_w_id IN (1,2,4)
) OR (
  ol_i_id = i_id
  AND i_data LIKE '%c'
  AND ol_quantity >= 1
  AND ol_quantity <= 10
  AND i_price BETWEEN 1 AND 400000
  AND ol_w_id IN (1,5,3)
);
```

Query 19 suits very well to our purpose, it contains 15 predicates, and it contains both Boolean operators: AND, OR. Further, it contains predicates with varying costs, including cheap predicates with only comparison operators (i.e., $\leq, \geq$), somewhat more expensive predicates containing the `IN, BETWEEN` clauses, and expensive predicates containing the `LIKE` clause.

Since we do not rely on the IA, selectivity estimates for all subsets of predicates have to be supplied to our algorithms. For this purpose, we have used the sampling method shown in Section 8.1 over the materialized relation obtained as a product of joining `item` and `orderline` (on attributes `ol_i_id = i_id`). The newly materialized relation contained 15 million tuples.

The results of this experiment are shown in Table 7.12. The middle column shows the overall plan costs (in seconds), and the right-most column shows the optimization time (in ms). The results of CNFALG are not shown as this algorithm did not terminate even after 10 minutes of optimization time. The

Algorithm	Plan costs (s)	Opt. time (ms)
BypassPlanGen	0.8	1.77
DNFalg	8.56	0.057
DNFdp	1.1	0.208

Table 6.12.: CH-benchmark results for Query 19

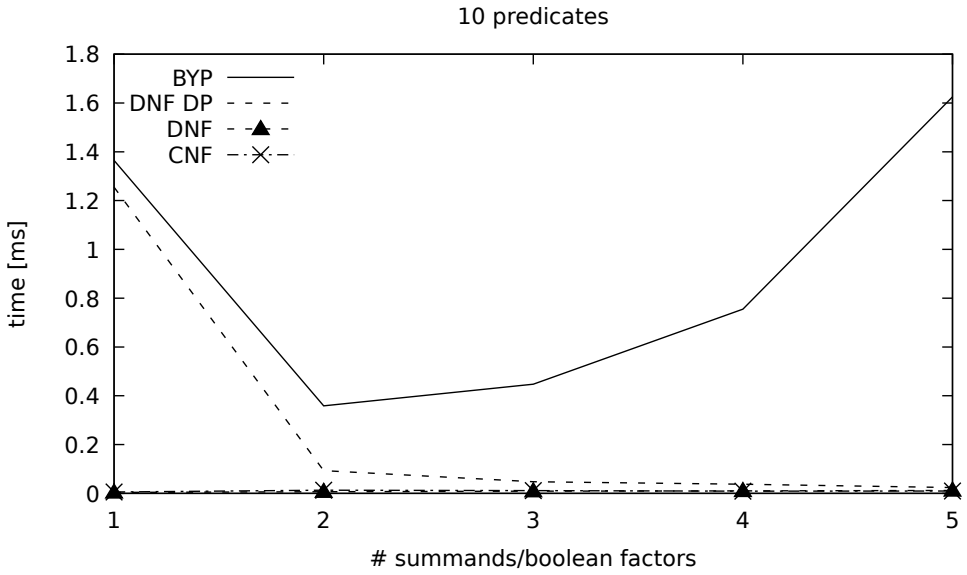


Figure 6.11.: The evaluation results of runtime performance

reason for this, is that Q19 is in DNF and had to be transformed into CNF, which in turn resulted with a very large query containing in total 125 Boolean factors, and each factor containing 5 predicates.

BypassPlanGen algorithm produced the best results; the plan produced by this algorithm is 27% cheaper than that of DNFdp, and 90% cheaper than that of DNFalg.

6.6.4. Runtime

Our experiments have convincingly shown that BypassPlanGen is the algorithm of choice. We are now interested to know what is the price that we have to pay if we choose BypassPlanGen over the normal form based algorithms (i.e., DNFalg, DNFdp, CNFalg)? We answer this question by measuring the runtimes of all the four algorithms used in our experiments. For the runtime measurements, we have used queries with 10 predicates, and a varying number of Boolean summands/factors. The results of this experiment are depicted in Fig. 6.11, where the x-axis shows the number of Boolean summands, respectively Boolean factors (for CNFalg), and the y-axis shows the elapsed time in milliseconds.

For queries with a single Boolean summand, both BypassPlanGen and

DNFDP have almost identical runtimes; the minimal difference is due to BYPASSPLANGEN having a larger constant. The reason for such identical runtimes is that both BYPASSPLANGEN and DNFDP employ DPSEL for building plans for conjunctive predicates, and since for a single summand there are only conjunctions of predicates (no OR operations) then the whole optimization job is performed by DPSEL, thus both algorithms produce the same results. Note that the time complexity of DPSEL is proportional to $O(n2^n)$, where n denotes the number of predicates in a conjunctive query (i.e., predicates in a single Boolean summand).

For queries with 2 Boolean summands the runtimes drop significantly for both BYPASSPLANGEN and DNFDP. The reason for this drop in runtime is mainly due to DPSEL, as for queries with 2 Boolean summands the number of predicates n has been halved in each summand, and since DPSEL has exponential complexity, for a smaller n it has to do less, therefore the runtime has dropped significantly for both algorithms. Of course, BYPASSPLANGEN is more expensive than DNFDP, as BYPASSPLANGEN has to consider all the permutations of Boolean summands, in addition to building the false branches for each bypass selection operator.

For queries with 3, 4, and 5 Boolean summands, the optimization time of BYPASSPLANGEN increases as expected. This happens due to the permutations of Boolean summands in BYPASSPLANGEN, and now the number of Boolean summands has increased. For DNFDP, the situation has improved, as the number of conjunctive predicates have dropped even further with the increase of the number of Boolean summands.

The runtime of the traditional algorithms DNFALG and CNFALG remains low for all the number of Boolean summands/factors; their complexity is proportional to $O(n \log n)$.

This experiment has shown that even for queries with 10 predicates with up to 5 Boolean summands, the optimization time of BYPASSPLANGEN remains well under 1.7 ms, which is a little price to pay given the possible gains in plan quality that BYPASSPLANGEN generates. We have seen in our previous experiments that for disjunctive queries with only 6 predicates, BYPASSPLANGEN beats DNF-based algorithms by factors of over 330, and CNFALG by a factor of over 2000! Regardless if predicates are cheap or expensive to evaluate, the gap on plan qualities between BYPASSPLANGEN and the normal form based algorithms is huge. Further, BYPASSPLANGEN is the best algorithm regardless if the query is in DNF or CNF.

6.7. Conclusion

In this chapter, we presented a heuristic optimization algorithm for disjunctive queries that generates bypass plans for main memory database systems. The algorithm presented in this chapter does not rely on common assumptions such as independence and constant predicate costs assumption. Moreover, the algorithm supports both common subexpression elimination and branch misprediction optimization, as well as common predicate elimination technique.

We have experimentally shown that regardless of the query's normal form (CNF or DNF) or the presence of common predicates, the algorithm presented in this chapter produces far superior plans compared to the heuristics algorithms found in the literature and in commercial database systems. The algorithm requires, however, the queries in DNF.

7. Optimal Evaluation of Boolean Expressions

In the previous chapter, we have tackled the problem of optimizing Boolean expressions by means of a heuristics optimization algorithm which—as validated by the experiments in Chapter 6—produces by far better plans than the existing heuristics in the literature and commonly used in RDBMSs. The heuristic from the previous chapter, however, has some limitations; it expects queries in DNF form, it does not recognize Boolean implications, and further, it does not guarantee the optimum. In this chapter, we present a new optimization algorithm for Boolean expressions, which suffers from none of those limitations.

7.1. Introduction

The most prevalent way of optimizing Boolean expressions in RDBMs relies on query normalization into either conjunctive normal form (CNF) or disjunctive normal form (DNF), and then order the conjuncts and disjuncts respectively. In both CNF and DNF optimization schemes, the search space is limited to the granules of Boolean factors/summands and not atomic predicates, as shown in Chapter 6. To exacerbate the matter, query normalization produces queries that are exponentially blown up in size, thus making them expensive to even prohibitively expensive to evaluate. We have shown experimentally in Chapter 6 that CNF and DNF based evaluation methods in general produce very poor plans.

In this chapter, we present a top-down optimization algorithm for Boolean expressions which attains the optimality in terms of plan quality and exhibits impressive runtimes, hence it is applicable for any practical application. In contrast to bottom-up algorithms top-down algorithms are amenable to search strategies like *branch-and-bound pruning* due to their demand driven nature. The branch-and-bound search strategy enables top-down algorithms to efficiently curtail their search space, and thus yielding better performance. The algorithm presented in this chapter does not require any normalization of queries, and the granules of optimization are the atomic predicates allowing for a fine-grained global ordering of the atomic predicates and this way making the optimum attainable. In addition, the algorithm presented in this chapter derives tighter upper bounds by means of Boolean Difference Calculus (BDC) [36] in order to prune the search space more aggressively, which translates to further improvements in the performance of the algorithm. To this end, the branch-and-bound pruning combined with BDC allow for an improvement of performance by an average factor of more than 5.

Since the algorithm presented in this chapter yields optimal plans, we are able to measure the quality of the exiting heuristics relative to the optimum. Furthermore, we can also measure the entire optimization potential available by measuring the gap between the worst and the best possible plan.

Recognition of *Boolean implications* in a query can help significantly reduce the search space as well as the plan costs. Consider the following example query predicate $(A_1 = \text{'ABC'} \wedge A_2 < 20) \vee (A_1 = \text{'DEF'} \wedge A_3 \neq 100)$. If $A_1 = \text{'ABC'}$ is true, then it must be that $A_1 = \text{'DEF'} = \text{FALSE}$, thus reducing the query to $A_2 < 20$. The work in the literature does not consider Boolean implications, whereas the algorithm presented in this chapter does.

As shown in Chapter 6, when optimizing Boolean expressions, the true theoretical optimization potential cannot be achieved by means of traditional plans. We can, however, fill this gap by means of bypass processing as in Chapter 6. Recall that in bypass processing, selection operators split the input tuple stream into two disjoint output streams: the *true-stream* with tuples that satisfy the selection predicate and the *false-stream* with tuples that do not. Bypass processing is crucial in avoiding expensive predicates whenever the outcome of the query predicate can be determined by evaluating the less expensive ones.

We summarize our contributions presented in this chapter as follows:

1. we present an optimization algorithm for Boolean expressions that guarantees the optimality with respect to the plan quality, while at the same time exhibits impressive runtimes,

2. our algorithm has made it possible for the first time to measure the gap on plan quality between the existing heuristics and the optimum, as well as the entire optimization potential available between the worst and the best possible plan,

3. our algorithm optimizes the branch misprediction penalty, supports CSE (common subexpression elimination), and duplicate predicate elimination,

4. our algorithm does not require normalization of Boolean expressions and does rely neither on IA (independence assumption) nor CC (constant predicate costs).

The rest of the chapter is organized as follows. Section 7.2 presents the optimization algorithm, wheres Section 7.3 presents a heuristics algorithm based on Boolean difference calculus. Section 7.4 presents some details on Boolean expression representation and Section 7.5 shows the evaluation results.

7.2. The Optimization Algorithm

In the following, we present our top-down optimization algorithm for Boolean expressions in a step-wise fashion in order to make it easier to understand. Our optimization algorithm *guarantees* the optimum in terms of plan quality in all the variants presented in this section.

```
TDSIM(e, Bxp, stream)
    // Input: partail plan e
              a query predicate Bxp
              flag stream
    // Output: best plan
 1  bestcost = ∞
 2  bestplan = NULL
 3  for each p ∈ {getPredicates(Bxp)}
 4       e' = BUILDPLANS(p_i, e, stream)
 5       e+ = TDSIM(e', Bxp[p ← TRUE], TRUE)
 6       e− = TDSIM(e', Bxp[p ← FALSE], FALSE)
 7       cost = Cost(e+) + Cost(e−) + Cost(e')
 8       if bestplan == NULL or bestcost > cost
 9            bestplan = [e', e+, e−]
10            bestcost = cost
11  return bestplan
```

Figure 7.1.: Pseudocode for TDSIM

As an introduction to top-down optimization of Boolean expressions, we first present the "simplest" variant of our top-down algorithm – TDSIM. Its pseudocode is shown in Figure 7.1.

7.2.1. The Basic Idea

TDSIM iterates over all the predicates in the input Boolean expression. Each predicate is evaluated by a bypass selection operator, which, in turn, has two output branches (i.e., the true and the false branch). The algorithm builds the true and the false branch by invoking itself recursively. On each recursive invocation, the input Boolean expression is *simplified* by assigning to the current predicate either true or false depending on the branch being built. The recursive descent stops once there are no predicates left in the input Boolean expression, that is, the expression has been simplified to either true or false. The simplification of Boolean expressions is explained in more detail in the following subsection.

7.2.2. The Algorithm in Detail

The algorithm accepts as an input a partial plan (e), a Boolean expression (Bxp), as well as a Boolean flag *branch*. The *branch* flag simply indicates whether the invocation of the routine TDSIM will build on top of the true or on top of the false branch of the preceding bypass selection operator in e. Initially, TDSIM is called with a partial plan (e) consisting of only a scan operator, as all selection operators evaluating the predicates in the input Boolean expression (Bxp) are built on top of the scan operator. Further, for the initial call, the

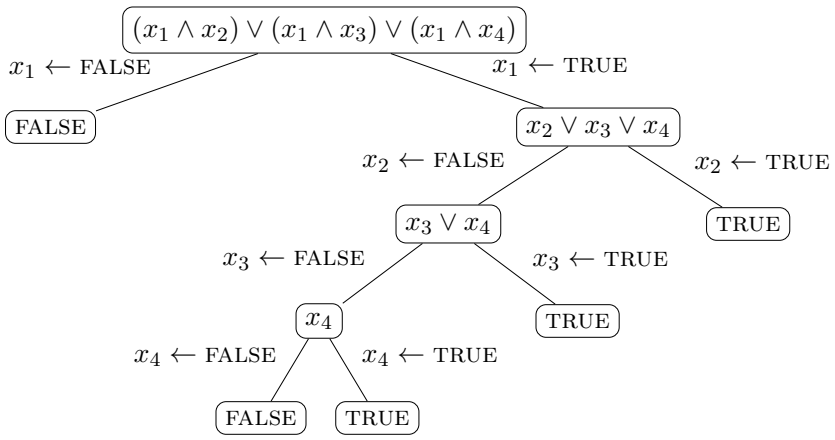


Figure 7.2.: Predicate assignment and Boolean expression simplification

branch flag is set to true; clearly the scan operator is not a bypass selection operator, therefore, it does not have a false branch.

In line 1, the variable *bestcost* is declared and initialized to ∞. In this variable, we store the cost of the best plan found in each invocation of the algorithm, whereas the best plan itself is kept in the variable *bestplan* (line 2).

In line 3, the algorithm iterates over the set of all predicates in the input Boolean expressions Bxp. Each predicate is evaluated by a single bypass selection operator, which is added to the partial plan e by means of the auxiliary method BuildPlans. The resulting partial plan returned from BuildPlans is kept in e' (cf. line 4).

Since each bypass selection operator has two branches, we need to build these branches, too, and repeat the same process until the plan evaluating the entire expression (Bxp) is built. Branches in turn are built by two recursive descents shown in lines 5-6. The true branch is kept in e^+, and the false branch in e^-. In each such recursive descent, the input Boolean expression is simplified by assigning to the current predicate p either a true ($p \leftarrow true$) or a false ($p \leftarrow false$) value, depending on the branch being built.

To illustrate the simplification of Boolean expressions, consider the following example expression:

$$(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_1 \wedge x_4).$$

Let us assume for the moment that the bypass selection operator for the predicate $p = x_1$ is built, and that the algorithm makes two recursive descents (cf. lines 5-6 of TDSim) in order to build its two branches. The assignments, as well as an instance of the resulting simplified expression, is depicted in Figure 7.2.

The simplification $x_1 \leftarrow$ FALSE reduces the entire expression to false, hence, the recursive descent on the false branch of σ^-_{x1} returns immediately, that is, the bypass selection operator evaluating x_1 has an empty false branch. Its true branch, however, will be built by the recursive invocations of the algorithm as explained, until the expression is simplified to either true or false.

Note that the assignments of truth values to predicates ensure *duplicate predicate elimination*. Whenever an assignment is made to a predicate p_i, all oc-

```
TDmemo(e, Bxp, stream)
    // Input: partail plan e
              a query predicate Bxp
              flag stream
    // Output: best plan
 1  if Memo[Bxp] ≠ NULL
 2       return Memo[Bxp]
 3  bestcost = ∞
 4  bestplan = NULL
 5  for each p ∈ {getPredicates(Bxp)}
 6       e' = BuildPlans(p_i, e, stream)
 7       e+ = TDmemo(e', Bxp[p ← TRUE], TRUE)
 8       e− = TDmemo(e', Bxp[p ← FALSE], FALSE)
 9       cost = Cost(e+) + Cost(e−) + Cost(e')
10       if bestplan == NULL or bestcost > cost
11            bestplan = [e', e+, e−]
12            bestcost = cost
13  Memo[Bxp] = bestplan
14  return Memo[Bxp]
```

Figure 7.3.: Pseudocode for TDmemo

currences of p_i in the Boolean expression are updated with the assigned truth value, resulting in a new reduced expression as illustrated in Figure 7.2. The simplification of Boolean expressions is proportional to $O(n)$ for n predicates in the expression, as the algorithm has to iterate over all the predicates in the expression.

In line 7, the algorithm computes the costs of the partial plan in e' including its true e^+ and its false e^- branch, respectively. In lines 8-10, the algorithm checks if the *bestplan* is null, or the newly found plan in e' is cheaper than the *bestplan* found so far. If any of these two conditions holds, the newly found plan together with its branches is kept in the *bestplan* variable. Once the iteration over the variables in the input Bxp is exhausted, the cheapest plan found (*bestplan*) is returned to its caller (cf. line 11).

New predicates are added to the exiting (partial) plans in the same fashion as in the heuristics algorithm shown in Section 6.5.2. That is, new predicates are appended to the existing (partial) plans by means of the BuildPlans procedure shown in Figure 6.7 (cf. Section 6.5.2). Finally, the newly constructed plan resulting from the invocation of the BuildPlans procedure is returned to the caller, i.e., to the TDsim algorithm.

7.2.3. Memoization

In the previous subsection, we showed the simplest variant of our top-down

optimization algorithms for Boolean expressions. We now present an improved version of TDSIM coined TDMEMO with its pseudo code shown in Figure 7.3.

In this approach, the recomputation of the solutions for the recurring (sub)expressions is avoided by means of memoization. That is, once a partial plan for an expression has been computed, it gets registered (i.e., memoized) in an associative data structure – *memo table*. Whenever the same expression recurs, we fetch the already computed plan from the table associated with that particular expression and thus avoid the expensive recomputation. Since subexpressions often recur, memoization yields large improvements in the performance of the algorithm—by an average factor of over 200—as we will show in Section 7.5.

Assignments. Boolean expressions cannot be used directly as keys for the memo table in the actual code. For this purpose we have devised an *assignment structure* composed of two bitvectors. The first bitvector (`predicates_bv`) represents the predicates which were assigned truth values, whereas the second one (`values_bv`) represents the truth values assigned to the predicates, i.e., 1 or 0 in bitwise representation. Now each time we need to associate or fetch a plan for a particular Boolean expression in the memo table, we can efficiently compute the hash code for the expression by means of the assignment structure, and then use the computed hash code for querying the memo table. Thanks to the assignment structure, hash codes for Boolean expressions can be very efficiently computed. They require only two bitwise operations, namely a left shift on the predicates bitvector and a bitwise-OR over the two bitvectors:

```
int hashcode() {
  return ((predicates_bv << 32) | (values_bv));
}
```

In the pseudocode shown above, we are assuming that the bitvectors are stored in integral integer types of width 64 bits. In the pseudocode that follows, we will directly use Boolean expressions as keys for the memo table, but it should be understood that in reality, we use their hash codes generated by means of their assignment structures. Note that for every Boolean expression with n predicates, there are at most 3^n different assignments possible.

Line 1 of TDMEMO checks against the memo table if the plan for the input expression (Bxp) has already been computed, and if so, in line 2 the plan is fetched from the table and returned to the caller. If otherwise, the plan has to be computed in the same fashion as in the TDSIM algorithm, but in contrast to TDSIM, before the newly computed plan is returned, it gets first registered in the memo table (cf. line 13).

7.2.4. Branch-and-bound Pruning

As we have seen with TDSIM and TDMEMO, the nature of our top-down optimization algorithms is demand-driven. For each bypass selection operator, its branches (true and false) are built on request. The demand driven nature allows for employing the *branch-and-bound* pruning technique. The beauty of this technique is that it enables the algorithm to reduce significantly its search

space without jeopardizing optimality. In the worst case, the algorithm gracefully degrades into exhaustive search, e.g., like TDSIM.

Branch-and-bound pruning is a state-of-the-art search strategy exploited in the join optimization domain for successfully reducing the search space, as shown in [19, 20]. We present in the following subsection the first optimization algorithm for Boolean expressions that applies branch-and-bound pruning search strategy. To the best of our knowledge, this is the first time that this search strategy is being applied to the problem of optimization of Boolean expressions. The experiments presented in Section 7.5 show that branch-and-bound pruning yields large factors of improvement in the performance of the algorithm.

7.2.5. Accumulated-Cost Bounding

Accumulated-cost bounding works by passing down a cost budget to the top-down optimization procedure, while each recursive invocation of the optimization procedure subtracts costs from the handed-over budget as soon as they become known. The recursive descent halts once the budget drops below zero. Notable systems implementing this technique in the realm of join optimization are Volcano [23], Cascades [22] and Columbia [59].

The pseudocode of our algorithm —TDACB— implementing this technique is shown in Figure 7.4. Line 1, just as in TDMEMO, checks if the plan for the input expression has been already computed and if that is the case, it also makes sure that its cost does not exceed the handed-over budget (b) before returning it. The condition in line 3, on the other hand, checks if the budget is lower than the already known lower bound (LB) for the input expression. If this condition holds, it is fruitless to continue with the plan construction, as it will not become a part of the optimal plan. Lower bounds are maintained in the memo table – LB. If the lower bound for an expression is not set, LB returns 0 by default.

Accumulated-cost bounding is a very efficient technique in preventing the algorithm to build non-promising branches, which will not constitute the final plan. However, in some scenarios, this technique can in fact make the algorithm more expensive than the one without it, e.g., TDMEMO. This behavior has also been observed in the join optimization domain [20]. Such worst case behavior occurs whenever a partial plan for some expression is requested a number of times, and each time the handed-over budget is only slightly higher than before, but still too low to produce the cheapest partial plan. This in turn leads to the cascading negative effect of unnecessary computations of partial plans on each such request. We solve this problem by proposing a *rising budget*. In line 5, we check if the lower bound associated with the input expression is greater than 0, and if so, we know that a plan for the same expression has been requested before, but was not constructed due to the insufficient budget. For this reason, the budget is increased to the double of the current lower bound—given that the handed-over budget is not greater than that—as shown in line 6 of TDACB. The doubling of the budget alleviates the negative cascading effect of unnecessary multiple plan computations due to the insufficient budget.

In line 9, the algorithm checks if the budget is sufficient for exploring the

```
TDACB(e, Bxp, stream, b)
    // Input: partail plan e, query predicate Bxp
              flag stream, cost budget b
    // Output: best plan
 1  if Memo[Bxp] ≠ NULL and Cost(Memo[Bxp]) ≤ b
 2      return Memo[Bxp]
 3  if LB[Bxp] ≥ b
 4      return NULL
 5  if LB[Bxp] > 0
 6      b = MAX(b, LB[Bxp] * 2)
 7  bestcost = ∞
 8  bestplan = NULL
 9  if b ≥ 0
10      for each p ∈ {getPredicates(Bxp)}
11          e' = BUILDPLANS(p_i, e, stream)
12          b' = MIN(b, bestcost) − Cost(e')
13          e+ = TD_ACB(e', Bxp[p ← TRUE], TRUE, b')
14          if e+ ≠ NULL
15              b' = b' − Cost(e+)
16              e− = TD_ACB(e', Bxp[p ← FALSE], FALSE, b')
17              if e− ≠ NULL
18                  cost = Cost(e+) + Cost(e−) + Cost(e')
19                  if bestplan == NULL or bestcost > cost
20                      bestplan = [e', e+, e−]
21                      bestcost = cost
22  // If no valid plan was found with budget b
23  if bestplan.e+ == NULL or bestplan.e− == NULL
24      LB[Bxp] = b
25      return NULL
26  Memo[Bxp] = bestplan
27  return Memo[Bxp]
```

Figure 7.4.: Pseudocode for TDACB

search space further. If that is not the case, the recursive descent stops. If otherwise, in line 12, the budget is updated, where the costs of the newly built partial plan e' are subtracted from the budget. The updated budget is stored in the new variable b'. Line 13 requests the true branch of e'. This request is made by the recursive invocation of TDACB with the updated budget b'. If no plan is returned, it becomes clear that the handed-over budget was insufficient, hence the partial plan e' cannot become a part of the final plan. If otherwise, in line 16 the algorithm requests the false branch of e', but this time with even a tighter budget. If the request succeeds, that is, a plan is returned, then in similar fashion as with TDSIM and TDMEMO, in lines 19-21 we check if the new plan is cheaper than the one stored in *bestplan*, and if so, we keep the new plan in the *bestplan* variable.

If no valid plan[1] was found, we set the lower bound for expression Bxp to the handed-over budget (line 24). If a plan is requested again for the same expression, and the handed-over budget lies below the lower bound set for the expression, we can spare the efforts in constructing the plan as we know that the budget will not suffice. That is, such recurring request will immediately terminate, as shown in lines 3-4. If, on the other hand, a valid plan was found, we register it with the memo table and return it to the caller. Note that for the initial call, TDACB is handed a budget set to ∞.

To this end, the cost of the plan produced by means of Boolean difference calculus (BDC) can be effectively leveraged for setting the *upper bound* in our top-down optimization algorithm. So instead of calling TDACB with the initial budget set to $b = \infty$, we set the budget to the cost of a plan built by means of BDC. By reducing the initial budget the algorithm can prune the search space more aggressively, which translates to a better performance (cf. Section 7.5). Details of a heuristic algorithm based on BDC are given in Section 7.3. The variant of our top-down algorithm which works with a budget set by means of BDC will be referred to as TDACB w. BDC.

The time complexity of our top-down algorithm is $O(n3^n)$, where n is the number of predicates in the Boolean expression. The original input Boolean expression sets the upper bound in terms of the number of possible assignments that can be made to the expression, that is, the upper bound is proportional to 3^n. The upper bound can be derived from the fact that a predicate can be assigned true/false value or left unassigned.

The complexity of our algorithm is far lower than the algorithm by Reinwald and Soland's which also achieves the optimum, but has a time complexity proportional to $(O(2^{2^n}))$, for more details on this algorithm see Section 6.2 of Chapter 6. Further, due to memoization and applying intelligent search techniques like branch-and-bound, together with efficient upper bound derivation by means of BDC, TDACB outperforms TDSIM (which resembles the worst case behavior) and TDMEMO by large margins, as shown experimentally in Sec. 7.5.

[1] An invalid plan is a plan with one or both empty branches (i.e., branches set to null).

7.2.6. Predicted-Cost Bounding

Accumulated-cost bounding works by passing the budget information in top-down fashion, where the budget plays the role of the upper bound. Predicted-cost bounding works by predicting what lies ahead; for a plan e', we can predict the lower bound by *estimating* the costs of its true e^+ and false e^- branch without building the actual partial plans through recursive descents. If the lower bound turns out to be larger than the cost of the already found best plan for the given Bxp, we know that e^+ and e^- cannot be part of the optimal plan and, hence, we can spare the recursive descents.

Predicted-cost bounding is not new, it has been successfully used in join enumeration algorithms [19, 20]. A notable system implementing this technique for join optimization is the Columbia [59]. In contrast to the join optimization problem, this technique proved unsuccessful for optimizing Boolean expressions. The only way to efficiently derive the lower bound estimates is by taking the costs of evaluating the cheapest outstanding predicate in each of the two branches. This, however, yields a rather weak bound (overly conservative) which does not help much in pruning, but only in increasing the costs due to the additional computation overhead incurred by the lower bound computation. To this end, we have decided to not keep this technique in our algorithm, as based on our experiments it proved unsuccessful.

7.2.7. Boolean Implications

The recognition of Boolean implication can have a great impact on the plan quality as well as on the reduction of the optimization time. Let us consider Q19 of the CH-Benchmark [62]:

```
SELECT  SUM(ol_amount) AS revenue
FROM  orderline, item
WHERE (
  ol_i_id = i_id
  AND i_data LIKE '%a'
  AND ol_quantity >= 1
  AND ol_quantity <= 10
  AND i_price BETWEEN 1 AND 400000
  AND ol_w_id IN (1,2,3)
) OR (
  ol_i_id = i_id
  AND i_data LIKE '%b'
  AND ol_quantity >= 1
  AND ol_quantity <= 10
  AND i_price BETWEEN 1 AND 400000
  AND ol_w_id IN (1,2,4)
) OR (
  ol_i_id = i_id
  AND i_data LIKE '%c'
  AND ol_quantity >= 1
```

```
  AND ol_quantity <= 10
  AND i_price BETWEEN 1 AND 400000
  AND ol_w_id IN (1,5,3)
);
```

If the predicate `i_date LIKE '%a'` is true, then it implies that the predicates `i_date LIKE '%b'`, and `i_date LIKE '%c'` are false, thus reducing the entire query to the first clause only. Boolean implications are recognized during the simplification of the expression ($p \leftarrow$ TRUE/FALSE) in our algorithm, e.g., see lines 5,6 of Figure 7.1. Boolean implications help in reducing the optimization time significantly, due to the reduced query size, and moreover, they yield cheaper evaluation plans. The effect of Boolean implication on the optimization time is confirmed by our experiments in Section 7.5.6 and Section 7.5.6.

7.3. Boolean Difference Calculus

Boolean difference calculus (BDC) is a heuristic which has been used for Boolean expression optimization in [36]. It works by ordering the predicates based on their influence on the outcome of the expression. The predicate with the highest influence is picked first, and then its true/false branches are built recursively, where the succeeding predicates are chosen in the same fashion. The influence (or the rank) of the predicates is computed by means of *Boolean difference*:

$$\begin{aligned}\Delta_{x_i} f(x_1, .., x_{i-1}, x_{x+1}, .., x_n) \stackrel{\text{def}}{=} \\ f(x_1, .., x_{i-1}, x_i = \text{FALSE}, x_{i+1}, .., x_n) \not\equiv \\ f(x_1, .., x_{i-1}, x_i = \text{TRUE}, x_{i+1}, .., x_n)\end{aligned}$$

The higher the probability (i.e., selectivity) $s(\Delta_{x_i} f)$ of $\Delta_{x_i} f$ being true, the higher is the influence of the predicate x_i on the outcome of the Boolean expression. Having defined the Boolean difference, the ranks of predicates can be computed according to the equation:

$$rank_{x_i} = s(\Delta_{x_i} f)/c(x_i),$$

where $c(\cdot)$ denotes the predicate evaluation cost. The pseudocode of the BDC-based algorithm is shown in Fig. 7.5. Its time complexity is $O(n^2)$ for n predicates.

BDC does not guarantee the plan optimality, however, in our context, it can be used efficiently for decreasing the initial budget passed to our top-down optimization algorithm TDACB.

7.4. Boolean Expression Implementation

Boolean expressions are commonly represented as a tree of nodes, where each node corresponds to a conjunct ($\wedge$), or a disjunct ($\vee$), or an atomic predicate. Nodes can have zero or more children whereby the pointers to children nodes

```
BDC(e, Bxp, stream)
    // Input: partail plan e
              a query predicate Bxp
              flag stream
    // Output: best plan
 1  bestplan = NULL
 2  bestvar = ∞
 3  bestrank = 0
 4  for each p ∈ {getPredicates(Bxp)}
 5       if rank(p) > bestrank
 6            bestvar = p
 7            bestrank = rank(p)
 8  e' = BuildPlans(bestvar, e, stream)
 9  e+ = BDC(e', Bxp[bestvar ← TRUE], TRUE)
10  e− = BDC(e', Bxp[bestvar ← FALSE], FALSE)
11  bestplan = [e', e+, e−]
12  return bestplan
```

Figure 7.5.: Pseudocode for BDC

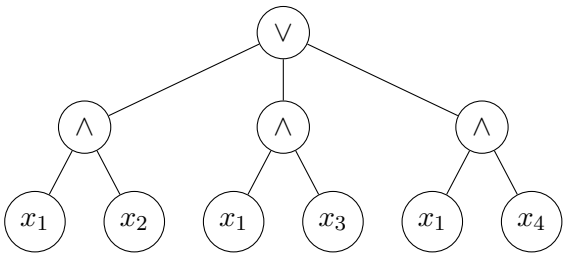


Figure 7.6.: Tree representation of the expression $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_1 \wedge x_4)$

are kept in an array pointers. An example depicting the tree representation for a Boolean expression is given in Figure 7.6.

Tree implementation is straightforward and allows for easy traversal, as well as manipulation of the tree, e.g., adding/removing nodes. However, the main drawback with such a tree implementation are its high costs, coming as a result of pointer chasing. In order to fulfill standard operations such as reading or deleting nodes, we have to follow the corresponding pointers leading to the desired nodes, which turned out to be a costly operation as the nodes are scattered throughout the memory, thus reducing the cache locality. Furthermore, in such a representation, each new node requires memory allocation, and memory allocations are relatively slow operations.

We thought we could do better, therefore we decided to encode expressions in a very compact form – in bitvector representation. We allocate in one operation a contiguous memory chunk, large enough to hold all the bitvectors required to encode the input Boolean expression. Each bitvector corresponds to an integral

# Pred.	TDsim (bitvector)	TDsim (tree)
4	0.012	0.43
6	0.376	11.8
8	22	351
10	2192	22940

Table 7.1.: Tree vs. bitvector representation (runtimes in ms)

data type such as unsigned integer, having a width of 32 or 64 bits, depending on the architecture. In a single bitvector we encode a single Boolean connective $(\wedge, \vee)$, together with its children, which can be atomic predicates or simply indexes to other bitvectors (Boolean connectives) in the allocated memory chunk. Bitvector representation of expressions yields a very small memory footprint, which in turn fits nicely into the CPU cache. Moreover, a single memory chunk allocation is a much more efficient operation than allocating each node separately as it is the case with the tree representation.

In order to show the overhead incurred by the tree representation in our optimization algorithm (versus bitvector representation) we have performed a small experiment. Queries used in this experiment were in CNF form and contained two atomic predicates in each Boolean factor. As a test algorithm we have picked TDsim. That is, we ran TDsim with queries in tree representation and bitvector representation. The recorded runtimes of this experiment are shown in Table 7.1. The left column shows the total number of predicates used in the experiment, the middle column shows the execution time of TDsim with queries in bitvector representation, and the right column shows the execution time of TDsim with queries in tree representation. The experiment was run single-threaded, on a machine with Intel Xeon E5-2690 v2 3.00GHz processor. The machine had 120 GB of main memory, running a 64-bit linux operating system.

As it can be seen in Table 7.1, the same optimization algorithm running with queries in bitvector representation yields much lower execution times than when using tree representation. The difference in execution times gets as large as a factor of over 30, thus showing the large overhead incurred when processing expressions in tree representation.

7.5. Evaluation

Two features of optimization are of high significance in query optimization: (1) the execution costs of the optimized query, i.e., the plan quality, and (2) the costs incurred by the optimization algorithm.

We first measure the execution costs incurred by our top-down algorithms, and then compare the winner against two state-of-the-art heuristics for Boolean expression optimization.

All experiments were run single-threaded on a machine with an Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz processor. The machine was equipped

with 264 GB of main memory, and ran a 64 bit Arch Linux operating system. All the algorithms were implemented in `C++`, and compiled using `g++` (version 6.2.1).

7.5.1. Forest dataset

We have already introduced the Forest [14] in the evaluation sections of Chapter 5 and Chapter 6. In this section we will use the same dataset to test the algorithms introduced in this chapter. Recall that the Forest dataset contains data about US forests, materialized in a relation with 54 attributes, and 581.012 tuples. This rather wide relation validates the importance of optimizing Boolean expressions.

For testing our algorithms we have used a set containing in total 16 predicates. The predicates were simple range predicates of the form $c_1 \leq attr_i \leq c_2$, were c_1, c_2 denote integer constants. As we do not rely on the independence assumption, selectivity estimates for all the subsets of predicates were derived by means of the very efficient sampling method shown in Chapter 8.

We compared all algorithms by generating randomly $10k$ queries over randomly chosen attributes of the Forest relation. For all experiments, we used queries in both normal forms: DNF and CNF. This way, we did not give any unfair advantage to any algorithm over the others. Furthermore, most queries we see in practice are either in CNF or DNF with deeper nested queries being quite rare.

7.5.2. Runtime of Top-Down Algorithms

Table 7.2 and Table 7.3 compare the runtimes of our top-down optimization algorithms, including their *minimum*, *average* as well as their *maximum* runtime.

For this experiment, each Boolean summand/factor consisted of two atomic predicates. TDSIM was tested only up to a maximum of 10 predicates, due to its prohibitively large runtimes. Starting with 6 predicates, TDSIM was outperformed by TDMEMO by an average factor of over 2. For 8 predicates the gap reached a factor of 18, and for 10 predicates a factor of over 200, thus showing the significance of memoization in our top-down algorithm.

Branch-and-bound pruning with accumulated-cost bounding (TDACB) showed very promising results. Even for a small number of predicates, e.g., 6 predicates, TDACB outperforms TDMEMO by 20%. The gap kept increasing steadily with the increase of the number of predicates, irrespective of the query type (DNF or CNF). For 16 predicates and queries in DNF, the gap between TDACB and TDMEMO got as large as a factor of 4, whereas for CNF queries the factor was 2.7.

As shown in subsection 7.2.6, Boolean Difference Calculus (BDC) can be useful in setting the upper bound in TDACB, and this way allowing for better search space pruning. BDC showed improvements in TDACB for queries with more than 12 predicates. For queries in DNF with 14 predicates (cf. Table 7.3), BDC helped improve the average runtime of TDACB by 17%, and for 16 predicates by 37%. For queries in CNF, the improvements were not so strong; for

# pred.	TDsim			TDmemo			TDacb			TDacb w. BDC		
	min	avg	max	min	avg	max	min	avg	max	min	avg	max
4	0.009	0.012	0.029	0.014	0.02	0.045	0.007	0.017	0.048	0.009	0.017	0.047
6	0.33	0.376	0.77	0.15	0.16	0.38	0.029	0.127	0.366	0.037	0.133	0.372
8	21	22	50	0.96	1.17	3.02	0.187	0.748	2.9	0.165	0.741	2.9
10	2171	2192	2355	9.8	10.2	26	0.73	6.4	24	0.823	6.5	24
12				98	114	172	11.6	57	140	12.2	57	134
14				1457	1495	1565	141	637	1789	133	621	1781
16				13587	13734	13906	1193	5104	14122	1144	4987	14007

Table 7.2.: Performance (in ms) for CNF query type $(p_1 \vee p_2) \wedge \ldots$

# pred.	TDsim			TDmemo			TDacb			TDacb w. BDC		
	min	avg	max	min	avg	max	min	avg	max	min	avg	max
4	0.01	0.012	0.026	0.014	0.019	0.065	0.007	0.02	0.043	0.009	0.02	0.056
6	0.35	0.386	0.727	0.155	0.175	0.374	0.04	0.16	0.383	0.046	0.167	0.406
8	19	20	48	0.991	1.1	2.8	0.088	0.905	3.3	0.097	0.9	3.4
10	2109	2124	2261	10.1	10.6	27	0.674	7.1	26	0.717	7.1	27
12				101	118	190	1.32	55	127	1.6	56	119
14				1391	1536	1587	1.26	515	1469	2.7	440	1478
16				14255	14336	14520	135	3452	9662	65	2514	9509

Table 7.3.: Performance (in ms) for DNF query type $(p_1 \wedge p_2) \vee \ldots$

# pred.	TDsim			TDmemo			TDacb			TDacb w. BDC		
	min	avg	max	min	avg	max	min	avg	max	min	avg	max
4	0.007	0.011	0.033	0.011	0.015	0.038	0.007	0.014	0.033	0.008	0.016	0.038
6	0.276	0.309	0.64	0.112	0.13	0.292	0.016	0.068	0.234	0.023	0.071	0.245
8	16.5	16.9	43.3	0.78	0.9	2.4	0.077	0.445	2.083	0.096	0.443	2.1
10	1587	1599	1677	6.4	6.8	19.05	0.277	2.8	13.7	0.343	2.9	14.2
12				74.8	76.9	153	2.1	21.1	56.4	2.5	21.4	56.1
14				1025	1039	1142	23.5	234	805	25.1	223	805
16				8733	8833	9016	298	1900	5604	275	1774	5564

Table 7.4.: Performance (in ms) for CNF query type $(p_1 \vee p_2 \vee p_3) \wedge \ldots$

# pred.	TDsim			TDmemo			TDacb			TDacb w. BDC		
	min	avg	max	min	avg	max	min	avg	max	min	avg	max
4	0.007	0.01	0.024	0.01	0.015	0.042	0.01	0.016	0.037	0.009	0.017	0.04
6	0.285	0.318	0.644	0.114	0.135	0.279	0.063	0.124	0.3	0.07	0.129	0.312
8	17	17.4	42.8	0.795	0.989	2.4	0.384	0.931	2.5	0.398	0.935	2.5
10	1618	1633	1724	6.6	6.9	18.4	0.923	5.01	15.1	1.072	5.055	15.3
12				77.2	78.7	147	35.1	61.1	118	35.3	61.2	118
14				1045	1059	1251	265	665	1068	165	640	1056
16				9281	9339	9573	279	3712	8810	6.5	2721	8726

Table 7.5.: Performance (in ms) for DNF query type $(p_1 \wedge p_2 \wedge p_3) \vee \ldots$

14 and 16 predicates, BDC helped decrease the average runtime of TDACB by 2.5%.

We repeated the same experiment, but this time we increased the number of OR respectively AND operations in each Boolean summand/factor. That is, each Boolean summand/factor now contains 3 atomic predicates. As before, we used queries with 4 and up to 16 predicates. For those numbers of predicates that could not be perfectly divided by 3, we appended the expression with an additional summand/factor containing the remainder of predicates, e.g., for 8 predicates we have the following (DNF) expression: $(p_1 \wedge p_2 \wedge p_3) \vee (p_4 \wedge p_5 \wedge p_6) \vee (p_7 \wedge p_8)$, where all the Boolean summands have three conjuncts with the exception of the last one, which has only two. The results of this experiment are shown in Tables 7.4 and 7.5.

As in the first experiment, branch-and-bound pruning strategy yields significant improvements in the runtime of our top-down algorithm (TDACB). For 6 predicates and CNF queries, TDACB outperforms on average TDMEMO by 47% , whereas the latter outperforms TDSIM by 57%. With the increase of the number of predicates, the gaps on performance between TDACB and TDMEMO increased steadily, such that for 16 predicates, TDACB beats TDMEMO by a large factor of over 4. BDC, same as before, helped further improve the runtime of TDACB for queries with more than 12 predicates; for 14 predicates, BDC helped reduce the average runtime of TDACB by 4.7% and for 16 predicates by 6.6%.

For queries in DNF, the gaps between TDACB and TDMEMO are not as large as for queries in CNF, but nevertheless significantly large. For all the number of predicates TDACB outperformed TDMEMO; for 16 predicates TDACB outperformed TDMEMO by a factor of 2.5, whereas BDC helped further improve the performance of TDACB by as much as 26%.

To summarize, our experiments have shown that memoization is instrumental in reducing the runtime of our top-down algorithm; the gap on the average runtime between the algorithm without memoization (TDSIM) and the one with memoization (TDMEMO) gets as large as a factor of 200 for only 10 predicates (cf. Table 7.3). Further, the branch-and-bound search strategy showed significant improvements in all the queries. The gap on performance is as high as a factor of 4 on average for queries with 16 predicates relative to the algorithm not implementing this search strategy – TDMEMO.

7.5.3. Evaluation of two heuristics

In this subsection we present the results of evaluating two existing state-of-the art heuristics-based optimization algorithms in terms of plans quality against the optimum, i.e., against the plans produced by our TD algorithms.

The heuristics we used are the following:

1. The heuristics algorithm presented in Chapter 6 which relies on a permutation of Boolean summands. We abbreviate this algorithm in the rest of this thesis writing as PBS.

# pred.	BDC		PBS		Opt. pot.	
	avg	max	avg	max	avg	max
4	1.15	6.98	1.01	1.12	1.86	18.17
6	1.18	3.93	1.01	1.11	2.12	17.81
8	1.23	3.52	1.01	1.16	2.67	25.14
10	1.26	5.10	1.01	1.13	3.26	24.27
12	2.76	22.06	1.02	1.11	8.76	45.24
14	3.60	61.43	1.03	1.24	13.01	135.97
16	4.08	81.28	1.84	24.58	16.64	198.51

Table 7.6.: Performance of the heuristics against the optimum for the Forest dataset for DNF queries: $(p_1 \wedge p_2) \vee \ldots$

# pred.	BDC		PBS		Opt. pot.	
	avg	max	avg	max	avg	max
4	1.16	4.63	1.19	1.80	1.86	13.30
6	1.19	5.38	1.97	2.97	2.30	16.14
8	1.22	5.00			2.76	26.42
10	1.24	5.39			3.02	18.81
12	1.36	5.10			2.07	9.98
14	1.36	4.53			2.16	18.37
16	1.36	5.31			2.24	17.88

Table 7.7.: Performance of the heuristics against the optimum for the Forest dataset for CNF queries: $(p_1 \vee p_2) \wedge \ldots$

# pred.	BDC		PBS		Opt. pot.	
	avg	max	avg	max	avg	max
4	1.10	7.72	1.01	1.15	2.61	64.38
6	1.25	7.16	1.01	1.15	2.55	34.33
8	1.20	3.69	1.01	1.09	2.48	17.98
10	1.14	8.71	1.01	1.07	5.02	52.70
12	1.69	14.18	1.02	1.10	5.05	30.13
14	2.83	60.19	1.61	24.44	8.39	134.75
16	7.74	211.86	2.62	50.80	24.36	498.92

Table 7.8.: Performance of the heuristics against the optimum for the Forest dataset for DNF queries: $(p_1 \wedge p_2 \wedge p_3) \vee \ldots$

# pred.	BDC		PBS		Opt. pot.	
	avg	max	avg	max	avg	max
4	1.14	89.03	1.46	95.61	2.94	119.83
6	1.24	67.04	1.29	51.83	2.36	149.66
8	1.23	74.06			2.41	158.53
10	1.20	3.24			3.47	33.63
12	1.81	10.40			2.94	14.67
14	1.73	9.26			2.81	13.49
16	1.58	8.45			3.00	55.13

Table 7.9.: Performance of the heuristics against the optimum for the Forest dataset for CNF queries: $(p_1 \vee p_2 \vee p_3) \wedge \ldots$

2. A heuristics algorithm based on BDC [36] with its pseudo code shown in Section 7.3.

An evaluation of the heuristics for Boolean expressions was not possible before, as the existing algorithms in the literature that attain the optimum cannot handle more than a few predicates due to their prohibitively high runtime (cf. Section 6.2).

For this experiment, we have used predicates with varying costs over the Forest dataset. That is, we have assigned random cost values to the map operators, uniformly distributed in the range [1, 100]. For each query, there were 100 such random cost assignments to the map operators that predicates depended on.

The results of this experiment are shown in the Table 7.6 – Table 7.9. The *average* as well as the *maximum* deviation factor(!) of the heuristics against the optimum in terms of plan quality is shown.

For DNF queries with two atomic predicates in each conjunct and 4 predicates in total, BDC deviates from the optimum on average by a factor of 1.15 (cf. Table 7.6), whereas its maximum deviation is a factor of 6.9. The gaps on the plan quality increase with the increase of the number of predicates. For 16 predicates, the average deviation from the optimum of plans produced by BDC is a factor of 4, whereas its maximum deviation factor is 81! Plans produced by PBS, for 4 predicates, deviate on average from the optimum by factor of 1.01, whereas their maximum deviation from the optimum is a factor of 1.12. For 16 predicates, the average and the maximum deviation of plans produced by PBS reach factors of 1.8 and 24 respectively. On average PBS produced better plans than BDC over the entire range of queries, further, PBS proved to be more robust, that is, its worst case behavior is better than that of BDC. Nevertheless, both heuristics can be far off the optimum.

For CNF queries with binary predicates in each Boolean factor (cf. Table 7.7), plans produced by BDC deviate on average from the optimum by an average factor of 1.16, and in the worst case by a factor of 4.3. PBS requires queries in DNF, therefore, it could not handle queries in CNF with more that 6 predicates, due to the exponential query blow-up in size, which is a consequence of converting the queries from one normal form to another.

Table 7.8–Table 7.9 show the results of our experiments with an increased number of AND (for DNF queries) respectively OR (for CNF queries) operations. For DNF queries with 4 predicates, BDC produced plans that on average deviate from the optimum by a factor of 1.1, and the gap keeps increasing with the increase of the number of predicates, this way reaching a factor of 7.7 for 16 predicates. The gaps on maximum deviation are far worse, they start from a factor of 7.7 for 4 predicates and reach a factor of over 200 for 16 predicates! PBS on the other hand, has again produced better plans in contrast to BDC; for 4 predicates, plans produced by PBS deviate on average from the optimum by a factor of 1.01, and for 16 predicates the factor on deviation gets as large as 2.6. In the worst case, plans produced by PBS deviate from the optimum by a large factor of 50.

For CNF queries with ternary atomic predicates in each Boolean factor, the results for the heuristics were not that impressive, plans produced by BDC deviated from the optimum by as much as a factor of 89, whereas PBS deviated by a large factor of 95.

7.5.4. Optimization Potential

Next, we show the entire optimization potential available for the queries used in these experiments. Since we can compute the optimum by means of our top-down algorithm(s), we can also compute the entire optimization potential available, which is the space between the best possible plan (optimum) and the worst possible plan. The worst possible plan can be easily computed by any variant of our top-down algorithms by simply altering the statement $bestcost > cost$ to $bestcost < cost$ (cf. line 19 of Figure 7.4).

In Table 7.6 and Table 7.7 we show the maximum optimization potential for DNF and CNF queries with binary predicates in each Boolean summand/factor. The optimization potential is shown in the column '`Opt. pot.`'.

The optimization potential greatly increases with an increasing number of predicates. On average, for DNF queries, the gap between the optimum and the worst plan for 4 predicates is a factor of 1.86 and quickly reaches a factor of 16 for 16 predicates. If we take the maximum, the gap reaches a factor of 198! For CNF queries, the average gap between the optimum and the worst plan is bounded by a factor of 2.7, whereas the maximum deviation consists of a factor of 26.

For DNF queries with ternary predicates in Boolean summands, the average gap between the optimum and the worst plan is a factor of 24, whereas the maximum is a factor of 498 (cf. Table 7.8). For the CNF counterpart, the average deviation between the best and the worst possible plan was bounded by a factor of 3 and the maximum deviation factor by 158 (cf. Table 7.8). To this end, such a large optimization potential confirms the importance of optimizing Boolean expressions.

# pred.	$(p_1 \wedge p_2) \vee \ldots$			$(p_1 \wedge p_2 \wedge p_3) \vee \ldots$		
	BDC	TDACB w. BDC	PBS	BDC	TDACB w. BDC	PBS
4	0.003	0.02	0.01	0.003	0.02	0.01
6	0.01	0.167	0.05	0.01	0.13	0.03
8	0.03	0.9	0.3	0.03	0.93	0.06
10	0.09	7.1	2.43	0.09	5.06	0.28
12	0.4	56	31	0.47	61	1.93
14	1.83	440	457	2.11	640	7.15
16	9.06	2514	8011	7.79	2721	87

Table 7.10.: Runtimes for DNF queries (in ms)

# pred.	$(p_1 \vee p_2) \wedge \ldots$			$(p_1 \vee p_2 \vee p_3) \wedge \ldots$		
	BDC	TDACB w. BDC	PBS	BDC	TDACB w. BDC	PBS
4	0.003	0.017	0.08	0.002	0.01	0.07
6	0.01	0.133	74.15	0.005	0.07	228
8	0.03	0.741		0.01	0.44	
10	0.10	6.5		0.06	2.9	
12	0.41	57		0.42	21.4	
14	1.91	621		2.00	223	
16	9.44	4987		6.98	1774	

Table 7.11.: Runtimes for CNF queries (in ms)

7.5.5. Runtime

In the previous subsection, we have seen that the gap between the heuristics and the optimum is quite large. We are now interested in answering the question: what is the cost of applying our algorithm which attains the optimum (e.g., TDACB) instead of heuristics? We answer this question by measuring the performance of our most efficient top-down algorithm – TDACB w. BDC (abbreviated in the rest of this section as only TDACB), and the two heuristics. We have recorded the average optimization time for all the algorithms over $10k$ queries over the Forest dataset. The results of this experiment for both DNF and CNF queries are shown in Table 7.10 and Table 7.11. All the results show the optimization time required by the algorithms in milliseconds (ms).

BDC has quite a low runtime due to its low complexity (cf. Section 7.3). PBS has lower runtimes than TDACB only up to 12 predicates for DNF queries, for more than 12 predicates TDACB outperforms it. For 16 predicates, TDACB beats PBS by a factor of 3, while at the same time guaranteeing the optimum. For CNF queries, PBS cannot handle more than 6 predicates, while TDACB and BDC do not have this limitation.

7.5.6. CH-benchmark

In this section we show the results of comparing the algorithms by using the CH-benchmark [13, 62] workload. CH-benchmark is a complex, mixed workload

Algorithm	Plan costs (s)	Opt. time (ms)
TDACB	0.6	5.2
TDACB (B. impl.)	0.6	2.7
TDACB w. BDC	0.6	5.1
TDACB w. BDC (B. impl.)	0.6	2.8
BDC	0.7	0.1
PBS	0.8	1.8
Worst plan	8.3	5.2

Table 7.12.: CH-benchmark results for Query 19

benchmark, devised with the goal of closing the gap between TPC-C (for OLTP) and TPC-H (for OLAP). For the purpose of testing our algorithms, we have picked Query 19 of this benchmark (shown in Section 7.2.7).

Query 19 suits our purpose very well, it contains 15 predicates, and both boolean operators: AND, OR. Further, it contains predicates with varying costs, including cheap predicates with only comparison operators (i.e., $\leq, \geq$), somewhat more expensive predicates containing the `IN, BETWEEN` clauses, and expensive predicates containing the `LIKE` clause.

Since we do not rely on the IA, selectivity estimates for all subsets of predicates have to be supplied to our algorithms. For this purpose, we have used the sampling method shown in Chapter 8.1 over the materialized relation obtained as a product of joining `item` and `orderline` (on attributes `ol_i_id = i_id`). The newly materialized relation contained 15 million tuples.

The results of this experiment are shown in Table 7.12. The left column lists the algorithms tested in this experiment while the middle column shows the overall plan costs (in seconds), and the rightmost column shows the optimization time (in ms). TDACB has produced the cheapest plan of all the algorithms, i.e., the optimum. The plan produced by BDC is 16.6% more expensive than the optimum, and the plan produced by PBS is 33.3% more expensive than the optimum. In Table 7.12 we have also shown the effect of *Boolean implications* (entries marked with "B. impl.") in the optimization time. The recognition of Boolean implications has roughly halved the optimization time of our algorithm (TDACB) without sacrificing plan optimality.

TDACB with cost bounding based on BDC (TDACB w. BDC) when considering Boolean implications has resulted in an optimization time slightly higher than that of TDACB without BDC. The reason for this is that Boolean implication has reduced Q19 to a single clause only (i.e., to a query with only 5 predicates instead of 15), which is a rather small query, therefore the overhead of computing the initial cost bound by means of BDC has shown its effect. The opposite was true when the Boolean implications were not considered, TDACB w. BDC performed slightly better than TDACB.

The last row in Table 7.12 shows the worst possible plan for Q19, which in turn gives us a clue for the large optimization potential available for this query, i.e., a factor of 14.

#ibxp	S	#var	no Boolean impl. Opt. time [ms]	with Boolean impl. Opt. time [ms]
2	0	8	1.807	1.81
2	1	7	0.72	0.72
2	2	6	0.33	0.32
3	0	12	116.7	112
3	1	10	22.1	20.5
3	2	8	2.99	2.74
4	0	16	8111	6250
4	1	13	393.9	332
4	2	10	29.8	24.5

Table 7.13.: The effect of Boolean implications on CNF queries

#ibxp	S	#var	no Boolean impl. Opt. time [ms]	with Boolean impl. Opt. time [ms]
2	0	8	1.48	1.24
2	1	7	0.77	0.55
2	2	6	0.28	0.25
3	0	12	110.2	59.29
3	1	10	21.8	10.37
3	2	8	3.08	1.46
4	0	16	8376	2695
4	1	13	378.7	157.6
4	2	10	28.02	12.02

Table 7.14.: The effect of Boolean implications on DNF queries

7.5.7. Boolean Implications

To better see the effect of Boolean implication in the optimization time we have performed another experiment. We have used queries in both conjunctive (CNF) and disjunctive normal forms (DNF) over the Forest dataset. Queries were optimized by our top-down optimization algorithm – TDMEMO. The results of this experiment are show in Table 7.13 and Table 7.14, whereby:

- `#ibxp`: denotes the number of Boolean factors (for CNF queries) respectively the number of Boolean summands (for DNF queries),
- `S`: denotes the number of shared (atomic) predicates, and
- `var`: denotes the total number of (atomic) predicates given in the query.

In the first set of the experiments the optimizer ran with Boolean implications recognizer turned off. The results of this experiment are shown under the column "no Boolean impl.". In the second set of experiments we enabled the Boolean implication recognizer. The results of this experiment are shown under the column "with Boolean impl.".

# pred.	BDC		PBS		Opt. pot.	
	avg	max	avg	max	avg	max
3	1.11	8.95	1.01	1.15	1.39	25.81
4	1.15	9.32	1.01	1.18	1.60	26.19
5	1.18	29.83	1.01	1.19	2.03	43.07
6	1.19	10.89	1.01	1.19	1.72	29.33
7	1.27	14.21	1.01	1.18	2.28	37.09

Table 7.15.: Performance of the heuristics against the optimum for the Weather dataset for DNF queries: $(p_1 \wedge p_2) \vee \ldots$

As it can be seen in Table 7.13 and Table 7.14, recognition of Boolean implications helped reduced the optimization time in the vast majority of queries. That is, as our experiment shows, recognition of Boolean implications can help reduce the optimization time by a factor of over 3.

7.5.8. Weather Dataset

In this section, we show the results of our experiments over the Weather [43] dataset. The Weather dataset contains weather measurements for a single year, materialized in a relation with 7 attributes, and containing well over 3.4 million tuples. For this experiment, we have used a range of predicates starting from 3 up to 7 predicates in total. The predicates were simple range predicates of the form $c_1 \leq attr_i \leq c_2$, were c_1, c_2 denote constants. We compared all the algorithms by generating randomly $10k$ queries over randomly chosen attributes of the Weather relation. We have used predicates with varying costs over the Weather dataset, that is, we have assigned 100 different random cost values to the map operators for each query, where the cost values were uniformly distributed in the range [1, 100].

Just as in Sec. 7.5.4, we have compared the two heuristics against our top-down algorithm. Additionally, we show the entire optimization potential (Opt. pot.) available for the queries used in this experiment. Note that PBS for CNF queries was tested only up to 6 predicates as it cannot handle larger CNF queries due to the exponential blow-up in size when converting CNF queries to DNF. Recall that PBS requires that queries are first normalized in DNF before optimizing them, for more details see Chapter 6.

The results of this experiment are shown in Table 7.15 – Table 7.18. The results show the deviation (in factors!) of the heuristics in terms of plan quality relative to the optimum, i.e., against the plans produced by our top-down algorithm.

7.6. Conclusion

We have presented the first optimization algorithm for Boolean expressions that attains the optimum in terms of plan quality and has a much lower time complexity compared to the existing algorithms in the literature that attain the

# pred.	BDC		PBS		Opt. pot.	
	avg	max	avg	max	avg	max
3	1.07	15.08	1.51	1.99	2.35	25.79
4	1.15	7.31	1.15	1.83	1.63	26.46
5	1.07	12.48	1.89	7.24	4.18	42.69
6	1.19	6.18	1.54	1.95	1.54	23.44
7	1.01	13.14			6.31	54.42

Table 7.16.: Performance of the heuristics against the optimum for the Weather dataset for CNF queries: $(p_1 \vee p_2) \wedge \ldots$

# pred.	BDC		PBS		Opt. pot.	
	avg	max	avg	max	avg	max
3	1.44	24.95	1.01	1.10	2.66	25.82
4	1.11	12.14	1.01	1.15	2.10	40.15
5	1.17	7.55	1.02	1.20	1.72	28.47
6	1.21	9.75	1.01	1.16	1.77	26.66
7	1.22	3.91	1.01	1.13	1.62	17.69

Table 7.17.: Performance of the heuristics against the optimum for the Weather dataset for DNF queries: $(p_1 \wedge p_2 \wedge p_3) \vee \ldots$

# pred.	BDC		PBS		Opt. pot.	
	avg	max	avg	max	avg	max
3	1.21	36.69	1.01	1.15	2.04	25.80
4	1.05	9.53	1.68	2.45	3.65	40.15
5	1.17	12.25	1.7	1.82	1.82	26.37
6	1.26	15.54	1.57	1.69	2.96	29.33
7	1.01	5.08			5.95	52.84

Table 7.18.: Performance of the heuristics against the optimum for the Forest dataset for CNF queries: $(p_1 \vee p_2 \vee p_3) \wedge \ldots$

optimality. We have shown experimentally that search techniques like branch-and-bound with accumulated cost bounding together with memoization can help drastically reduce the runtime. Further, we have shown that recognition of Boolean implications in a query also helps significantly reduce the optimization time.

The impressive runtimes of our top-down optimization algorithm (TDAcb) have enabled us to measure the performance of the state-of-the-art heuristics algorithm against the optimum, and have seen that if we rely on heuristics for optimizing Boolean expressions, a large optimization potential remains unharvested. Such measurements wouldn't be feasible with the optimization algorithms from the literature which achieve the optimum.

7.6.1. Graceful Degradation

Generating optimal plans for Boolean expressions containing a large number of predicates (e.g., 100) might not be always feasible due to the prohibitively long computation times required by the optimization algorithm in presence of large expressions.

Dynamic programming/memoization optimization algorithms do not exhibit graceful degradation as the complete query execution plan is produced very late in the plan generation process. In our top-down approach we can, however, benefit from the Boolean difference calculus. That is, in TDAcb an initial plan is computed by means of BDC as its cost is used for setting the initial budget (cf. Section 7.3). In the presence of a large number of predicates in the input expressions, the algorithm can terminate at any time, and return the relatively good plan computed by means of BDC.

8. Cardinality Estimation

One cannot have a sound cost model without a sound framework on cardinality estimation as cost functions depend on cardinality estimates. It was shown in [38] experimentally, that the error in cardinality estimation dwarfs the error in the cost model.

The optimization algorithms presented in this thesis do not rely on the independence assumption as it typically does not hold [11], therefore, selectivity estimates for all the subsets of predicates given in a query have to be supplied. Selectivity estimates for subsets of predicates can be derived in several ways, e.g., by entropy maximization [42] or graphical models [63]. Both require some implementation effort and runtime. In this chapter, we present an easy to implement, and very efficient alternative. The main idea is to extend the usual sampling procedure to gather more than the usual information.

8.1. Cardinality Estimation based on Sampling

Let $P = \{p_1, \ldots, p_z\}$ denote a set of z predicates. For a subset of predicates $P' \subseteq P$, we denote by $\beta(P')$ the formula

$$\beta(P') = \bigwedge_{p_i \in P'} p_i, \tag{8.1}$$

and by $\gamma(P')$ the formula

$$\gamma(P') = \bigwedge_{p_i \in P'} p_i \wedge \bigwedge_{p_i \in P \wedge p_i \notin P'} \neg p_i. \tag{8.2}$$

The selectivities of these predicates are denoted by $s_{\beta(P')}$ and $s_{\gamma(P')}$. Thus, $\beta(P')$ is a conjunction of all predicates in P' whereas $\gamma(P')$ is a minterm for P.

As a technicality needed below, note that every subset $P' \subseteq P$ can be expressed as bitvector $\mathrm{bv}(P')$ of length $|P|$. Also, $\mathrm{bv}(P')$ can be interpreted as a positive integer whose representation it is. Subsequently, we will identify these two different interpretations of the same bitvector.

To illustrate the difference between s_γ and s_β, consider the sample data taken from the Forest [14] relation, shown in Table 8.1. Note that in Table 8.1 we are only showing a small sample consisting of only the first three attributes out of 54 attributes which the Forest relation contains. Now consider the following two predicates $p_1 = \mathit{elevation} > 2600$ and $p_2 = \mathit{slope} < 7$, and $P = \{p_1, p_2\}$. Values for both $s_{\beta(P)}$ and $s_{\gamma(P)}$ are shown in Table 8.2.

The left-most column of Table 8.2 shows the subsets P' of predicates in P in bitvector representation. The middle column shows the vector s_β, whereas the rightmost column shows the vector s_γ.

Elevation	Aspect	Slope
2596	51	3
2590	56	2
2804	139	9
2785	155	18
2595	45	2
2579	132	6
2606	45	7
2605	49	4
2617	45	9
2612	59	10

Table 8.1.: Sample data taken from the Forest [14] dataset

P'	$s_{\beta(P)}$	$s_{\gamma(P)}$
00	1	0
01	0.6	0.5
10	0.5	0.4
11	0.1	0.1

Table 8.2.: Values of $s_{\gamma(P)}$ and $s_{\gamma(P)}$ for the Forest sample

Let us briefly illustrate the difference in computation of selectivities for $\beta(P)$ and $\gamma(P)$. Lets take as an example the second row in Table 8.2 corresponding to the bitvector '01'. The selectivity $s_{\beta(P')}$ for $\text{bv}(P') = 01$ shows the selectivity of the predicate p_1 as only the first bit[1] in the bitvector is set to one (cf. Eq. 8.1). The same bitvector in case of $s_{\gamma(P')}$ has a different semantics; it shows the selectivity for the entire expression $p_1 \wedge \neg p_2$ and not only p_1 (cf. Eq. 8.2). That is, in case of $s_{\beta(P')}$, we compute the selectivities for predicates (interpreted conjunctively) corresponding to the bits set to one in the pattern P' whereas in case of $\gamma(P)$, all the bits in the pattern P' matter, regardless if they are set or not, since $\gamma(P')$ defines a minterm for P.

Since the computation of $\gamma(P')$ is little bit less intuitive than the computation of $\beta(P)$, we will illustrate it by using our example predicate expression $p_1 \wedge \neg p_2$. $\gamma(P')$ for this example expression is computed as the number of rows, where condition p_1 is true ($elevation > 2600$) and p_2 is false ($slope \geq 7$), and finally the derived count is divided by the total number of rows in the sample (i.e., 10, cf. Table 8.1). This gives us the value $\gamma(P) = 0.5$ as shown in Table 8.2.

Let us now discuss how to efficiently derive the values for s_γ via sampling. During the evaluation of a set of predicates $\{p_1, \ldots, p_z\}$, besides determining the number of sample tuples qualifying for all p_i, we can also count the 2^z combinations of predicates evaluating to true or false. In the following, we show the pseudocode (close to C++) of a method which efficiently achieves

[1] First bit in our interpretation is the least-significant bit

this task:

```
1  getGamma(p, z, S)
2    // p is vector of predicates,
3    // z its length,
4    // S is the sample
5    int n = (1 << z);
6    // array of counters initialized to zero
7    int c_gamma[n] = {0};
8    for(s : S) { // for all sample tuples in sample S
9      int k = 0;
10     for(int i = 0; i < z; ++i) {
11       // p[i](s): evaluate pi on sample tuple s
12       k |= (p[i](s) << i);
13     }
14     ++c_gamma[k];
15   }
16   double s_gamma[n];
17   for(int i = 0; i < n; ++i) {
18     s_gamma[i] = (double) c_gamma[i] / S.size();
19   }
20   return s_gamma;
```

Here, for every sample tuple $s \in S$, all predicates p_i are evaluated ($p[i](s)$). The result is either 0 or 1. Shifting this result by i and bitwise OR-ing it with k, stores this result in the i-th bit of k. Thus, after the inner loop (cf. lines 10-13), k contains a bitvector representing the outcome of all predicates. Then, k is used as an index into an array of counters and the according counter is increased.

8.2. Cardinality Estimation for Conjunctive Predicates

In Chapter 5, we have presented an optimization algorithm—DPSel—for queries containing predicates connected conjunctively. In order to compute plan costs, DPSel needs the vector s_β. Having s_β, the optimizer can quickly fetch predicate selectivities for any subset of conjunctive predicates directly from s_β.

The procedure `getGamma` presented in the previous section will give us s_γ, and not s_β. Hence, we need a method to convert s_γ to s_β.

Define the *complete design matrix* C as

$$C(i,j) = \begin{cases} 1 & \text{if } j \supseteq i \\ 0 & \text{else} \end{cases}$$

where $j \supseteq i$ denotes the fact that every bit set to one in i is also set in j, i.e., $i = i\&j$ and i, j range from 0 to $2^z - 1$. Note that C is binary, non-singular, and persymmetric.

The complete design matrix C allows us to go from s_γ to s_β by

$$Cs_\gamma = s_\beta.$$

Since the positions of the ones in row i can be enumerated efficiently by enumerating supersets of the bitvector i (see [44, p66] for details), multiplications

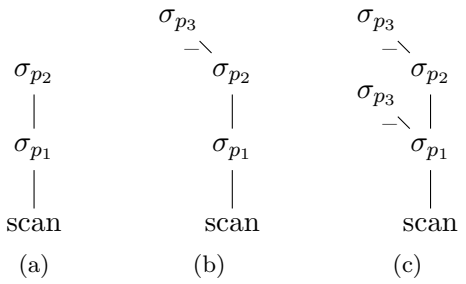


Figure 8.1.: An example bypass plan for expression $(p_1 \wedge p_2) \vee p_3$

of C with a vector x can be implemented very efficiently using only a few bit manipulating instructions and additions. No explicit representation of C is required.

8.3. Cardinality Estimation for Disjunctive Predicates

The optimization algorithms presented in this thesis, which handle general Boolean expressions (cf. Chapter 6 and Chapter 7), generate bypass plans. Recall that in bypass plans, each selection operator has two output streams, the true stream with the tuples satisfying the selection predicate and the false stream with tuples that do not satisfy the selection predicate. It follows that in order to compute the predicate selectivities required by our optimization algorithms generating bypass plans, we cannot rely on s_β as we have to also consider the false stream in each bypass selection operator. That is, besides conjuncts we need to compute selectivities for disjuncts too. For this purpose, we can use the vector s_γ which can be efficiently generated by our sampling method `getGamma`.

Once we have computed the vector s_γ by means of the sampling method shown in the previous section, cardinalities for bypass plans can be computed according to the method shown below (containing pseudocode close to C++).

```
1   calc_cardinality(A, nP, gamma)
2   // A  : assigment structure
3   // np : number of predicates
4   double res = 0;
5   if(0 == A.vars()) {
6     int n = (1 << np);
7     for(int i = 0; i < n; ++i) {
8       res += gamma[i];
9     }
10    return res;
11  }
12  int VarMask  = (1 << np) - 1;
13  int PartialAssignment = A.vars() & A.vals();
14  int Unassigned = ~(A.vars()) & VarMask;
15  // for all subsets (in bitvector rep.) in 'Unassigned'
16  for(s ⊆ Unassigned) {
17    res += gamma[PartialAssignment | s];
```

```
18    }
19    return res;
```

The method `calc_cardinality` takes as an input an assignment structure `A` (assignment structure was explained in Section 7.2.3), a variable `nP` denoting the number of predicates and a `gamma` vector (s_γ).

Recall that the family of top-down optimization algorithms presented in Chapter 7, iterate over the predicates in the input Boolean expression and recursively simplify the expression by assigning truth values to predicates, i.e., true/false. The assignment of truth values to predicates results with a simplified expression due to the cancellation of terms as a result of the assignment. An illustration of simplification of Boolean expressions has been shown in Figure 7.2 of Chapter 7. As already explained in Chapter 7, assignments made to predicates and their respective assigned values are maintained in an assignment structure.

In line 5 of the procedure `calc_cardinality`, it is checked if the assignment structure has no assigned predicates. If that results to be the case, the for loop in line 8 iterates over all the entries in vector s_γ, whereby their cumulative sum is stored in the variable res. If, however, the condition in line 6 is not satisfied, in lines 16-18 it is iterated over all the subsets $s \subseteq Unassigned$ (in bitvector representation). The variable $Unassigned$ denotes the set of predicates which are not assigned truth values (in bitvector representation), whereas the variable $PartialAssignment$ denotes the set of predicates which were assigned truth values. Having both $PartialAssignment$ and $Unassigned$ variables, all we need to do is iterate over all the subsets s of $Unassigned$ and take the union (bitwise-OR) of $PartialAssignment$ and s (cf. line 17) over the `gamma` vector and add the respective selectivity to the cumulative variable res. By iterating over all the subsets of predicates in $Unassigned$ (in bitvector representation), we have considered all the possible combinations of true/false values which predicates in $Unassigned$ can take. The variables $PartialAssignment$ and $Unassigned$ complement one-another in order to obtain the complete minterm which is necessary to query the `gamma` vector. Once the for loop in lines 16-18 has exited, the computed predicate selectivities are returned to the caller (cf. line 19).

For illustration of cardinality computation for bypass plans, consider the following example expression $(p_1 \wedge p_2) \vee p_3$. Let us assume that the optimizer has generated the bypass plan shown in Fig. 8.1a. For computing the input cardinality for the selection operator σ_{p_3} added on top of the false branch of σ_{p_2} as shown in Fig. 8.1b, we need the selectivity factor for the subset of predicates $P' = 101$ $(p_1 \wedge \neg p_2 \wedge p_3)$, whereas for the plan shown in Fig. 8.1c, we need the selectivity factor for the subset of predicates $P' = 110$ $(\neg p_1 \wedge p_2 \wedge p_3)$.

In order to complete the bypass plan, we need to build on the false branch of the bypass selection operator σ_{p_2} and this way derive the plan shown in Fig. 8.1b. For computing the input cardinality for the selection operator σ_{p_3}, the optimizer invokes the method `calc_cardinality` with the assignment structure A, whereby $A.vars = 011$ and $A.vals = 001$, $nP = 3$, and the vector s_γ. $A.vars$ denotes the already assigned predicates so far in the plan (in bitvector

representation). In contrast to the field $vars$, the field $vals$ encodes to the values assigned to each predicate in $A.vars$ (i.e., true/false). Note that the unassigned predicates have a *default* value of 0 in the assignment structure. When the optimizer considers building the false branch of the bypass selection operator σ_{p_2}, it has already modified the assignment structure by setting the bits corresponding to the predicates p_1 and p_2, hence $A.vars = 011$, and $A.val = 001$. Since we are building the false branch of the bypass selection operator σ_{p_2}, the predicate p_2 is assigned a false value, thus explaining $A.val = 001$. It follows that $PartialAssignment = 001$ and $Unassigned = 100$, thus in the res variable is stored the cumulative sum of selectivities for the indices $001, 101$ (in bitvector represenation) over the `gamma` vector.

9. Conclusion

The optimization algorithm presented by Ross in [56] is the only algorithm in the literature which optimizes conjunctive predicates for main memory databases, while taking into account the branch misprediction penalty. This algorithm, however, has a very high time complexity, proportional to $O(4^n)$ for n predicates. Moreover, the cost model in [56] is also too simple, i.e., it does not accurately capture the branch misprediction penalty. We have presented in Chapter 5, a new optimization algorithm for conjunctive predicates which relies on dynamic programming and generates the solutions in a bottom-up fashion. Its time complexity is much lower—$O(n\,2^n)$—compared to the algorithm by Ross [56], and in addition, in Chapter 4, we have presented a very accurate cost model for main memory column stores that accurately models the branch misprediction penalty.

The optimization of Boolean expressions which besides conjunctions contain disjunctions as well, is on the other hand, a much more challenging task. This class of queries has been traditionally optimized by very simple heuristics leaving a large optimization potential unharvested. In Chapter 6, we have presented a heuristic for optimizing this class of queries. The major drawback of our heuristic is that it requires queries in DNF, and while it performs very well for queries in DNF, it cannot handle large queries in CNF. The reason for that is the exponential blow-up of queries in size when converting them from one normal form to another normal form (e.g., CNF to DNF).

In Chapter 7, we have presented a top-down optimization algorithm for Boolean expressions which attains the optimum in terms of plan quality. The algorithm in Chapter 7 does not suffer from limitations of the heuristic in Chapter 6, and in addition, it also recognizes Boolean implications. The existing algorithms in the literature that attain the optimum have prohibitively high runtimes, thus are not applicable for queries with more than very few predicates. For instance, the algorithm by Reinwald and Soland [55] which attains the optimum in terms of plan quality has a complexity of $O(2^{2^n})$ for n predicates. In contrast, the complexity of our algorithm is much lower, i.e., $O(n\,3^n)$. Furthermore, our algorithm—thanks to its top-down nature—applies search strategies like branch-and-bound pruning for curtailing effectively its search space and Boolean difference calculus for setting the initial upper bound. Branch-and-bound pruning together with Boolean difference calculus improve the runtime of the algorithm by several orders of magnitude without jeopardizing the plan quality. For very large queries (e.g., containing over 100 predicates), however, computing the optimal plan is not feasible. Our algorithm in such cases can return a relatively good plan by means on Boolean difference calculus. As a future work, it would be interesting to develop other heuristics with the aim of closing the gap against the optimum.

A. Implementation Details

A.1. Allocator in SystemTx

```
class TxAllocator {
  public:
    static constexpr uint    Log2ChunkCardinality = 13;
    static constexpr size_t  ChunkCardinality = (1LL << Log2ChunkCardinality);
    static constexpr size_t  Mask = (ChunkCardinality - 1);
    static constexpr uint    Shift = Log2ChunkCardinality;
    typedef unsigned int flags_t;
    enum flags_et {
       kNoFlags = 0
    };
  public:
    typedef std::vector<void*> chunk_vt;
    constexpr static int64_t MINSIZE   = 16LL * 1024LL;
  private:
    TxAllocator(const TxAllocator&) = delete;
    TxAllocator& operator=(const TxAllocator&) = delete;
  public:
    TxAllocator();
    ~TxAllocator();
  public:
    void* alloc(const int64_t aElemSize, const flags_t aFlags);
  private:
    chunk_vt _chunks; // chunks
    void*    _begin;  // start off free space in current chunk
    void*    _end;    // end of current chunk
};

TxAllocator::TxAllocator()
            : _chunks() {
}

TxAllocator::~TxAllocator() {
  for(size_t i = 0; i < _chunks.size(); ++i) {
    free(_chunks[i]);
  }
}

void*
TxAllocator::alloc(const int64_t aElemSize, const flags_t aFlags) {
  int64_t lSize = (ChunkCardinality * aElemSize);

  if(lSize < MINSIZE) {
```

```
    lSize = MINSIZE;
  }

  void* lRes = 0;
  const int lRc = posix_memalign(&lRes, 64, lSize);
  if(lRc != 0) {
    return 0;
  }
  memset(lRes, 0x70707FAB, lSize);

  _chunks.push_back(lRes);

  return lRes;
}
```

A.1.1. Chunk-wise Column Organization in SystemTx

```
template<typename Tcontent>
class TxColumn {
  public:
    typedef Tcontent              elem_t;
    typedef std::vector<elem_t>   elem_vt;
    typedef std::vector<elem_t*>  elem_vpt;
  public:
    static constexpr size_t Mask  = TxAllocator::Mask;
    static constexpr uint   Shift = TxAllocator::Shift;
    TxColumn(const TxColumn&) = delete;
    TxColumn& operator=(const TxColumn&) = delete;
  public:
    TxColumn();
    ~TxColumn();
  public:
    void registerChunk(elem_t* aChunk);
    void initFrom(const elem_vt& aInitVec);
  public:
    inline size_t card() const { return _cardinality; }
    inline size_t cap()  const { return _capacity; }
    inline size_t noChunks() const { return _chunks.size(); }
    inline const elem_t* chunk(const size_t i) const { return _chunks[i]; }
  public:
    inline const elem_t& operator[](const size_t i) const;
    inline       elem_t& operator[](const size_t i);
  public:
    void push_back(const elem_t);
  private:
    size_t   _cardinality;
    size_t   _capacity;
    elem_vpt _chunks;
    size_t   _idx; // used for inserting elements iteratively
};
```

The following code snippet shows how column elements are accessed in our chunk-wise partitioned columns.

```
template<typename Tcontent>
typename TxColumn<Tcontent>::elem_t&
TxColumn<Tcontent>::operator[](const size_t i) {
  return _chunks[i >> Shift][i & Mask];
}
```

The variable `Shift` is a system parameter and fixed for the whole system (see the definition of `TxAllocator`). The operation `[i & Mask]` is a remainder operation and is equivalent to `[i % ChunkCardinality]`. Registration of the chunk pointers in a column is shown in the following code snippet.

```
template<typename Tcontent>
void
TxColumn<Tcontent>::registerChunk(elem_t* aChunk) {
  _capacity += TxAllocator::ChunkCardinality;
  _chunks.push_back(aChunk);
}
```

As it can be seen in the code snippet, chunks pointers are kept in the vector `_chunks`, further, each time a new chunk is registered, the cardinality of the column is updated accordingly. Note that the chunk cardinality and the column cardinality are two different things. The former denotes the number of items in a single chunk, whereas the latter denotes the total number of items in the column (covering all its constituent chunks).

Bibliography

[1] D Abadi, Daniel S Myers, David J DeWitt, and Samuel R Madden. Materialization strategies in a column-oriented dbms. In *ICDE*, pages 466–475. IEEE, 2007.

[2] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682. ACM, 2006.

[3] Anastassia Ailarnaki, David DeWitt, Mark Hill, and David Wood. DBMSs on modern processors: Where does time go? VLDB, 1999.

[4] José A Blakeley, William J McKenna, and Goetz Graefe. Experiences building the open oodb query optimizer. In *SIGMOD*, pages 287–296, 1993.

[5] Peter Boncz, Wilko Quak, and Martin Kersten. Monet and its geographical extensions: A novel approach to high performance gis processing. *Advances in Database TechnologyEDBT'96*, pages 145–166, 1996.

[6] Peter A Boncz, Stefan Manegold, and Martin L Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, volume 99, pages 54–65, 1999.

[7] Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, 2005.

[8] Jae-young Chang and Sang-goo Lee. An optimization of disjunctive queries: Union-pushdown. In *COMPSAC*, pages 356–361. IEEE, 1997.

[9] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards estimation error guarantees for distinct values. In *PODS*, pages 268–279, 2000.

[10] Surajit Chaudhuri, Prasanna Ganesan, and Sunita Sarawagi. Factorizing complex predicates in queries to exploit indexes. In *SIGMOD*, pages 361–372. ACM, 2003.

[11] Stavros Christodoulakis. Implications of certain assumptions in database performance evauation. *TODS*, pages 163–186, 1984.

[12] Jens Claussen, Alfons Kemper, Guido Moerkotte, Klaus Peithner, and Michael Steinbrunn. Optimization and evaluation of disjunctive queries. *Knowledge and Data Engineering, IEEE Transactions on*, 12(2):238–260, 2000.

[13] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, et al. The mixed workload ch-benchmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*, page 8. ACM, 2011.

[14] College of Natural Resources Colorado State University. Forest dataset. `http://kdd.ics.uci.edu/databases/covertype/covertype.data.html`.

[15] George P Copeland and Setrag N Khoshafian. A decomposition storage model. In *ACM SIGMOD Record*, volume 14, pages 268–279. ACM, 1985.

[16] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to algorithms*. MIT press Cambridge, 2001.

[17] G. Cormode, M. Garofalakis, P. Haas, and C. Jermaine. *Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches*. NOW Press, 2012.

[18] Transaction Processing Performance Council. TPC-H benchmark specification. *Published at http://www.tcp.org/hspec.html*, 2008.

[19] David DeHaan and Frank Wm Tompa. Optimal top-down join enumeration. In *SIGMOD*. ACM, 2007.

[20] Pit Fender, Guido Moerkotte, Thomas Neumann, and Viktor Leis. Effective and robust pruning for top-down join enumeration algorithms. In *ICDE*. IEEE, 2012.

[21] P. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *VLDB*, pages 541–550, 2001.

[22] Goetz Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3), 1995.

[23] Goetz Graefe and William J McKenna. The volcano optimizer generator: Extensibility and efficient search. In *IEEE Data Engineering*. IEEE, 1993.

[24] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. HYRISE: a main memory hybrid storage engine. *PVLDB*, pages 105–116, 2010.

[25] Michael Z Hanani. An optimal evaluation of boolean expressions in an online query system. *Communications of the ACM*, 20(5):344–347, 1977.

[26] Joseph M Hellerstein and Michael Stonebraker. *Predicate migration: Optimizing queries with expensive predicates*, volume 22. ACM, 1993.

[27] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[28] Toshihide Ibaraki and Tiko Kameda. On the optimal nesting order for computing n-relational joins. *TODS*, pages 482–502, 1984.

[29] Intel C++ intrinsic reference. `https://software.intel.com/sites/default/files/a6/22/18072-347603.pdf`.

[30] Matthias Jarke and Jurgen Koch. Query optimization in database systems. *ACM Computing surveys (CsUR)*, 16(2):111–152, 1984.

[31] Ryan Johnson, Vijayshankar Raman, Richard Sidle, and Garret Swart. Row-wise parallel predicate evaluation. *VLDB*, pages 622–634, 2008.

[32] Carl-Christian Kanne and Guido Moerkotte. Histograms reloaded: The merits of bucket diversity. In *SIGMOD*, pages 663–674, 2010.

[33] Fisnik Kastrati and Guido Moerkotte. Optimization of conjunctive predicates for main memory column stores. *VLDB*, pages 1125–1136, 2016.

[34] Fisnik Kastrati and Guido Moerkotte. Optimization of disjunctive predicates for main memory column stores. In *SIGMOD*, pages 731–744. ACM, 2017.

[35] Alfons Kemper, Guido Moerkotte, Klaus Peithner, and Michael Steinbrunn. Optimizing disjunctive queries with expensive predicates. In *ACM SIGMOD Record*, pages 336–347. ACM, 1994.

[36] Alfons Kemper, Guido Moerkotte, and Michael Steinbrunn. Optimizing boolean expressions in object bases. In *VLDB*, pages 79–90, 1992.

[37] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of nonrecursive queries. In *VLDB*, pages 128–137, 1986.

[38] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *VLDB*, pages 204–215, 2015.

[39] John DC Little, Katta G Murty, Dura W Sweeney, and Caroline Karel. An algorithm for the traveling salesman problem. *Operations research*, 11(6):972–989, 1963.

[40] Stefan Manegold, Peter Boncz, and Martin L Kersten. Generic database cost models for hierarchical memory systems. In *VLDB*, pages 191–202, 2002.

[41] V. Markl, G. Lohman, and V. Raman. LEO: An autonomic query optimizer for DB2. *IBM Systems Journal*, 42(1):98–106, 2003.

[42] Volker Markl, Peter J Haas, Marcel Kutsch, Nimrod Megiddo, Utkarsh Srivastava, and Tam Minh Tran. Consistent selectivity estimation via maximum entropy. *The VLDB journal*, 16(1):55–76, 2007.

[43] Matthew J Menne, Imke Durre, Russell S Vose, Byron E Gleason, and Tamara G Houston. An overview of the global historical climatology network-daily database. *Journal of Atmospheric and Oceanic Technology*, 29(7):897–910, 2012.

[44] G. Moerkotte. *Building Query Compiler*. 2014. pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf.

[45] Guido Moerkotte. Best approximation under a convex paranorm. `https://ub-madoc.bib.uni-mannheim.de/2279/1/MA_08_07.pdf`, 2008.

[46] Guido Moerkotte. *Building Query Compilers*. 2012.

[47] Guido Moerkotte, Martin Montag, Audrey Repetti, and Gabriele Steidl. Proximal operator of quotient functions with application to a feasibility problem in query optimization. *Journal of Computational and Applied Mathematics*, 285:243–255, 2015.

[48] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *VLDB*, pages 982–993, 2009.

[49] MOSEK. High performance software for large-scale LP, QP, SOCP, SDP and MIP.

[50] Kamesh Munagala, Shivnath Babu, Rajeev Motwani, and Jennifer Widom. The pipelined set cover problem. In *Database Theory-ICDT 2005*, pages 83–98. Springer, 2005.

[51] M Muralikrishna. Optimization of multiple-disjunct queries in a relational database system. Technical Report 750, University of Wisconsin, Madison, 1988.

[52] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[53] Thomas Neumann, Sven Helmer, and Guido Moerkotte. On the optimal ordering of maps and selections under factorization. In *ICDE*, pages 490–501, 2005.

[54] Holger Pirk, Florian Funke, Martin Grund, Thomas Neumann, Ulf Leser, Stefan Manegold, Alfons Kemper, and Martin Kersten. Cpu and cache efficient management of memory-resident databases. In *Data Engineering (ICDE)*, pages 14–25. IEEE, 2013.

[55] Lewis T Reinwald and Richard M Soland. Conversion of limited-entry decision tables to optimal computer programs I: Minimum average processing time. *Journal of the ACM (JACM)*, 13(3):339–358, 1966.

[56] Kenneth A Ross. Conjunctive selection conditions in main memory. In *SIGMOD*, pages 109–120, 2002.

[57] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34. ACM, 1979.

[58] S. Setzer, G. Steidl, T. Teuber, and G. Moerkotte. Approximation related to quotient functionals. *Journal of Approximation Theory*, pages 545–558, 2010.

[59] Leonard Shapiro, David Maier, Paul Benninghoff, Keith Billings, Yubo Fan, Kavita Hatwal, Quan Wang, Yu Zhang, H-M Wu, and Bennet Vance. Exploiting upper and lower bounds in top-down query optimization. In *IDEAS*. IEEE, 2001.

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

[61] David D Straube and M Tamer Özsu. Queries and query processing in object-oriented database systems. *ACM Transactions on Information Systems (TOIS)*, 8(4):387–430, 1990.

[62] TUM. CH-benchmark. `https://db.in.tum.de/research/projects/CHbenCHmark/`.

[63] K. Tzoumas, A. Deshpande, and C. Jensen. Efficiently adapting graphical models for selectivity estimation. *VLDB Journal*, 22:3–27, 2013.

[64] Marcin Żukowski. *Balancing vectorized query execution with bandwidth-optimized storage*. 2009.

[65] Marcin Zukowski, Peter A Boncz, Niels Nes, and Sándor Héman. Monetdb/x100-a dbms in the cpu cache. *IEEE Data Eng. Bull.*, 28(2):17–22, 2005.

[66] Marcin Zukowski, Mark Van de Wiel, and Peter Boncz. Vectorwise: A vectorized analytical dbms. In *ICDE*, pages 1349–1350, 2012.