Optimization of Conjunctive Predicates for Main Memory Column Stores

Fisnik Kastrati
University of Mannheim
Germany
kastrati@informatik.uni-mannheim.de

Guido Moerkotte
University of Mannheim
Germany
moerkotte@informatik.uni-mannheim.de

ABSTRACT

Optimization of queries with conjunctive predicates for main memory databases remains 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 work, we embrace 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.

1. INTRODUCTION

With the increase of main memory sizes as well as CPU cores per chip, and the decrease of their prices, main memory database systems are playing an ever increasing role in enterprise settings. Following this trend, a number of commercial main memory database systems have surfaced [8, 13, 30, 32, 36], in addition to a number of research-oriented prototypes [10, 15, 19].

By moving the database storage layer from disks to main memory, the performance improves drastically for data intensive applications. This move poses new challenges because details such as branch misprediction, which could easily be ignored in the context of disk-based systems, now rise to become prominent cost factors that can no longer be ignored.

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 [14] and in main memory, with the goal of achieving better query response times. In such settings, joins and I/O operations are not considered any longer the main cost [14], the evaluation of selection predicates has now taken the dominating role [14].

In this paper, 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. The goal is to give an optimization algorithm that determines the optimal evaluation order of selection predicates. As 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 [34, 37]. The second approach [11] orders predicates by *increasing* rank, where the rank of a predicate takes into account selectivities as well as costs and is defined as follows [12]:

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

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 [12]. 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 [4]. To see this, consider the beautiful example of Markl et al. [23]: `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 Fig. 4) and drop significantly if selectivities approach either 0 or 1. Since for

Proceedings of the VLDB Endowment, Vol. 9, No. 12

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 Sec. 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 Sec. 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 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. Thus, the evaluation of the expression p_1`&&`p_2 includes a conditional branch and, hence, 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.

Ross [31] has considered in detail the effect of conditional branches on plan quality, and offered 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 leaves a large optimization potential unharvested and calls for a new optimization algorithm that abandons both IA, CC and supports CSE. Further, the algorithm in [31] has a time complexity of $O(4^n)$. In contrast, the algorithm presented in this work has a much lower time complexity of $O(n2^n)$, while it does not rely on the IA or CC, and, in addition, it supports CSE.

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? We answer this question in Sec. 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. On the other hand, if for the QEE assumptions such as IA and CC do not hold, the QO must take that into account. Thus, the QO and the QEE very much depend on each other.

Summarizing, our contributions are as follows:

1. the first optimization algorithm neither relying on the IA (independence assumption) nor the CC (constant predicate costs assumption) and taking CSE (common subexpression elimination) into account, while supporting both branching-and(`&&`), and logical-and(`&`),

2. an error metrics together with a new theorem showing a direct link between cost estimation errors and plan quality,

3. precise cost functions exploiting recent advances in approximation theory, and

4. a sampling based method to compute the required selectivities in order to abandon IA.

The rest of the paper is organized as follows. Sec. 2 presents the related work. In Sec. 3 we present the preliminaries for this paper. Sec. 4 presents our error metrics as well as a theorem and a corollary linking cost estimation errors to plan quality. Sec. 5 introduces our cost model and Sec. 6 presents the optimization algorithm. Sec. 7 shows experimental results followed by Sec. 8, which concludes the paper. Appendix A presents the iterator model implemented in System Tx. Appendix B gives a very efficient technique to compute predicate selectivities based on sampling.

2. RELATED WORK

A number of commercial systems order predicates by increasing selectivity without consideration of their costs. A good example is Vectorwise [34, 37], a well-known column store geared for analytical workloads.

A more serious approach is presented by Hellerstein et al. [11]. They propose a scheme for ordering expensive predicates in an optimal way. Predicates can include non-trivial user defined functions (UDFs), that are expensive to evaluate. To this end, predicates are ranked in ascending order of the rank metric shown in the Eq. (1) in the introduction. This ranking metric originates from join-ordering [12, 20]. Hellerstein et al. [11] conclude that sorting of expensive predicates according to the above ranking metric produces the optimal plan. However, this is true only under the IA. We have already seen that this assumption does not hold. In their work, CSE is not considered, and in addition, predicate costs are assumed to be constant, i.e., they rely on the CC assumption, too.

Kemper et al. [18] consider optimizing boolean expressions in object databases by means of heuristics. However, their optimization scheme assumes both CC and IA. Moreover, CSE is not considered.

As mentioned in the introduction, Ross [31] considers the optimization of conjunction of simple selections over arrays

residing in main-memory, with the goal of optimizing the branch misprediction costs. In contrast to our work, the work in [31] does not provide *error bounds* and assumes that conjunctions are evaluated by a single operator over a complete materialization. His evaluation technique does not support CSE. Our algorithm in turn allows for pipelined query execution, where selection operators can be broken into a tree of operators. Furthermore, the algorithm in [31] relies on the IA and has a time complexity of $O(4^n)$. The time complexity of our algorithm is lower – $O(n\,2^n)$, while doing more: it considers CSE, and it does not rely on the IA.

The work by Munagala et al. [27] considers ordering of selection predicates by adopting approximation algorithms such as the set cover problem algorithm, coined *pipelined set cover*. The authors of [27] 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 processed does not provide an accurate cost function. Furthermore, this work relies on both constant predicate costs and the IA.

Neumann et al. [29] consider the optimization of selections depending on expensive UDF calls. Their work is based on both constant predicate costs and the IA.

Table 1 presents a clear view on related work, the assumptions they make, the support of CSE, and the support of branching(`&&`) vs. non-branching (`&`) code.

Work in	Assumes		Supports	
	IA	CC	CSE	(&&),(&)
Kemper et al.[18]	Yes	Yes	No	No
Hellerstein et al.[11]	Yes	Yes	No	No
Ross [31]	Yes	No	No	Yes
Munagala et al.[27]	Yes	Yes	No	No
Neumann et al.[29]	Yes	Yes	Yes	No
here	No	No	Yes	Yes

Table 1: Overview of related work

3. PRELIMINARIES

In this section, we present the algebraic operators used in this paper.

Sequential scan operator: `scan`(R)
This operator scans an input relation R by means of a tuple t. The tuple t contains an attribute named RID, which represents the row identifier (RID), 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 (see Appendix A).

Map operator: $\chi_{A_1:e'_1,\ldots,A_k:e'_k}(e)$ and $\chi_{*(A_1,A_2,\ldots,A_k)}(e)$
The map operator [2, 17] is of fundamental importance. It can extend a tuple produced by the input (partial) plan e, by a new attribute A whose value is calculated via an arbitrary expression e':

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

We generalize the map operator for many attributes as follows:

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

If we are interested not in the new attribute names, but only on the *dereference* (column access) operation, we denote the map operator by $\chi_{*(A_1,A_2,\ldots,A_k)}(e)$, where $A_1, A_2, \ldots, A_k$ stand for the attributes (i.e., columns) that this operator dereferences. The map operator in System Tx is used for dereferencing column values by means of either RIDs or column pointers (see Appendix A).

Selection operator: $\sigma_p(e)$ is the usual selection operator.

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*. 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

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

where

$$\|y\|_Q := \max\{y, 1/y\}.$$

Thus, the q-error measures the factor by which the estimate $\hat{x}$ deviates from the true value x. The q-error itself is well known [3, 7, 9, 16, 26], 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, 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} be the optimal plan for a query Q, minimizing $\mathcal{M}(e)$, and e_{best} the best plan, 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 4.1. *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 4.1 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 [26] 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. [21]. 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 4.2. *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 4.1 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[1]

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

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. 4.2 Assume $\mathcal{M}(e_{\text{best}}) \neq \mathcal{M}(e_{\text{opt}})$. Then, by Theorem 4.1 we have the following contradiction:

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

□

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

5. THE COST MODEL

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
P	set of predicates, interpreted conjunctively
sel(P)	selectivity of a set of 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
$\mathcal{M}(e)$	measured (true) cost, e.g., runtime for e

Table 2: Notation

$$\begin{aligned} \mathcal{C}(scan(R)) &= |R| * a_s + b_s \\ \mathcal{C}(\chi_{*(\mathcal{A})}(e)) &= |e| * (deref(1,n) + a_\chi) + b_\chi \\ \mathcal{C}(p_1 \& p_2) &= \mathcal{C}(p_1) + \mathcal{C}(p_2) + \mathcal{C}(\&) \\ \mathcal{C}(p_1 \&\& p_2) &= \mathcal{C}(p_1) + \mathcal{B}(s_1) + s_1\mathcal{C}(p2) \\ \mathcal{C}(\sigma_p(e)) &= |e| * (\mathcal{C}(p) + \mathcal{B}(sel(p)) + a_{in} + sel(p) * a_{out}) \end{aligned}$$

Table 3: Cost functions

This section contains our cost model as well as its validation. It is organized as follows. First, we state the basic cost functions for the physical operators scan, selection (σ), and map (χ) (see Sec. 3). 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 $p_1 \&\& p_2$. Afterwards, we present the cost functions for memory accesses and branch misprediction. Last, we evaluate the precision of our cost model.

As we will see, our cost functions 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. In our system, the calibration process is *automated*. Three plans are executed on different relations of varying sizes: (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. 5. The selection operator in System Tx 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.

Since we are interested in minimizing 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. Instead, we apply the approximation techniques presented in [33], since they allow approximations that directly minimize the q-error.

5.1 Cost Functions

For convenience, all notational details are summarized in Table 2, and all cost functions are presented in Table 3. Let us now briefly discuss the cost functions.

The scan and map operator both exhibit linear costs, and their cost functions are thus rather simple (see Table 3). These cost equations can be derived by looking at the implementation details of each operator (such details for System Tx are shown in Appendix A). For example, the scan operator depends on the relation size $|R|$ as well as constants, e.g., cost of incrementing iterator, tuple pipelining.

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

As noted before, 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 $p_1 \&\& p_2$. This explains the cost functions given in Table 3 for both of these expressions.

Last comes the selection operator. Its cost function 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.

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) $A_1, A_2, \ldots, A_k$.

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 Fig. 1. By subtracting the cost of the scan opera-

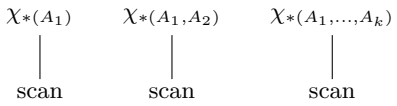

Figure 1: Plan types for measuring the costs of the dereference operator

tor, 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[2]:

$$\text{q-middle} = \sqrt{\max(x)\min(x)}$$

where x denotes the dereference costs. That is, we use a single constant for each number of simultaneously accessed columns.

The run-times for a number of plans dereferencing up to 9 different columns are shown in Fig. 2. The q-errors for all

[2]Also known as the geometric mean

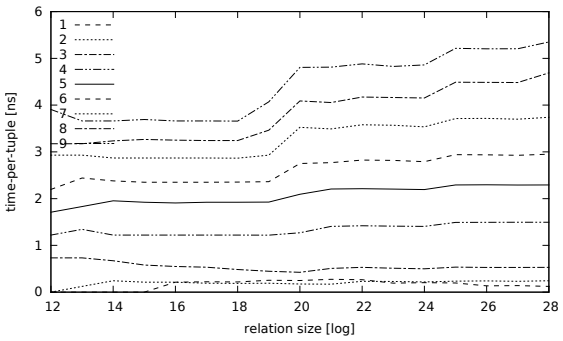

Figure 2: System Tx: column access costs

the plans and database sizes depicted in Fig. 2 are shown in Fig. 3. Note that we report the max q-error for *all* database sizes ($[2^{12}, 2^{28}]$), for up to 9 column dereferences at the time. The max q-error is very small for all the plans. In the worst

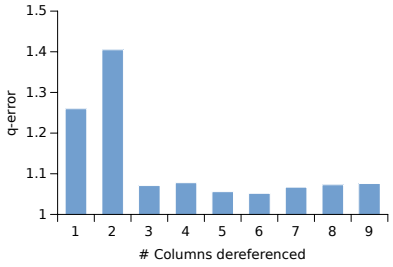

Figure 3: Q-Error of dereferenciation

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 for this is the hardware prefetcher.

5.3 Branch Misprediction Costs

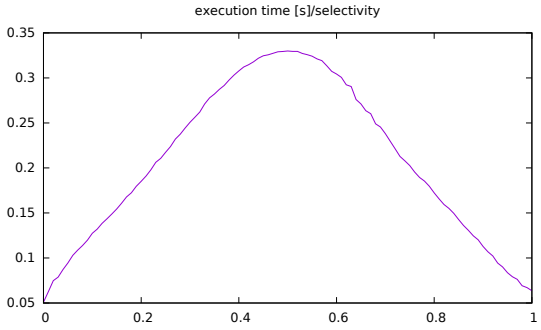

Figure 4: Execution time of a selection operator

Fig. 4 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.

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

$$\mathcal{C}(\sigma_p(A)) = n * (a_{in} + \mathcal{C}(p)) + n * s * a_{out} + n * \mathcal{B}(s), \quad (2)$$

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. (2) becomes

$$\mathcal{M}(s) = n * (a_{in} + \mathcal{C}(p)) + n * s * a_{out} + n * \mathcal{B}(s). \quad (3)$$

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. (3)

$$\mathcal{M}(s) = \mathcal{M}(0) + s * (\mathcal{M}(1) - \mathcal{M}(0)) + n * \mathcal{B}(s), \quad (4)$$

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

$$\mathcal{B}(s) = (\mathcal{M}(s) - \mathcal{M}(0) - s * (\mathcal{M}(1) - \mathcal{M}(0)))/n. \quad (5)$$

The branch misprediction can be very well approximated under the q-error [26] 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.

5.4 Cost Model Validation

In order to validate our cost model, we compared the measured execution times of several plans (see Fig. 5) with the execution times predicted by our cost model. These plan types 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. The q-error we report is the maximum over all these measurements. Table 4 shows the maximum q-error we observed for each of the plans in Fig. 5.

Table 4 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 QO's objective.

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: True vs. estimated costs

6. THE OPTIMIZATION ALGORITHM

In this section, we present our optimization algorithm coined DPSel. DPSel is responsible for producing query plans for evaluating conjunctions of selection predicates. It is based on dynamic programming. Fig. 7 shows its pseudocode.

DP algorithms generate solutions in a bottom-up fashion by combining solutions of smaller problems [6]. 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 maximation [22], or via sampling as shown in Appendix B. In addition, our devised cost model is utilized by DPsel to calculate actual costs. The output of DPsel is the best query evaluation plan, i.e., a plan with the lowest estimated execution cost. Thereby, DPsel requires neither the IA nor the CC assumption. Moreover, it supports CSE, branching-and(`&&`) and logical-and(`&`).

The algorithm starts by initializing an empty DP table and storing a plan consisting of only the scan operator (cf. lines 1-2 in Fig. 7). Operators evaluating selection 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 Fig. 6. This method 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* [29]. 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:

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

over some relation `R(A:int, B:text)`. Its dependency graph is shown in Fig. 9. 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 to find all those 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

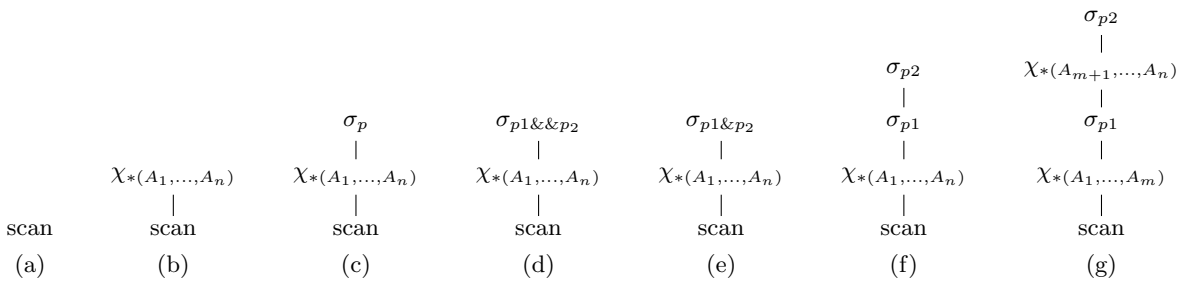

Figure 5: Plan types

BUILDPLANS(p, e)

Input: a selection predicate p
an expression e (partial plan)
Output: plan container B

1 $X_e = \cup_{p_i \in e} X_{p_i}$
2 $X_{p|e} = X_p \setminus X_e$ // outstanding maps
3 $B = \{\sigma_p(X_{p|e}(e))\}$
4 **if** $e == \sigma_{p'}(X_{p|e}(e'))$
5 $\quad B+=\sigma_{p'\&p}(X_{p|e}(e'))$
6 $\quad B+=\sigma_{p'\&\&p}(X_{p|e}(e'))$
7 **return** B

Figure 6: Pseudocode for BUILDPLANS

DPSEL

Input: a set $P = \{p_0, \ldots, p_{n-1}\}$ of predicates
Output: an optimal plan

1 DP = an empty DP table, size $\rightarrow 2^n$
2 $DP[\varnothing] = scan(R)$
3 **for each** $0 \leq i < 2^n - 1$ **ascending**
4 $\quad P' = \{p_k \in P \mid (\lfloor i/2^k \rfloor \bmod 2) = 1\}$
5 $\quad$ **for each** $p_j \in P \setminus P'$
6 $\quad\quad$ **for each** $e_j \in$ BUILDPLANS$(p_j, DP[P'])$
7 $\quad\quad\quad$ STORESOLUTION$(e_j, P' \cup \{p_j\}, DP)$
8 **return** $DP[P]$

Figure 7: Pseudocode for DPSEL

`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`. UDF function calls can be much more expensive to evaluate than column dereference operations, therefore considering CSE is of crucial importance when searching for the optimal plan.

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 $\vee\, \mathcal{C}(DP[P]) > \mathcal{C}(e)$
2 $\quad DP[P] = e$

Figure 8: Pseudocode for STORESOLUTION

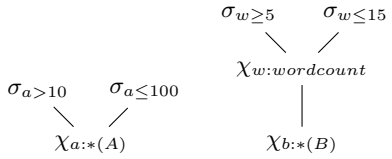

Figure 9: Dependency graph for the example query

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 Fig. 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. To this end, the time complexity of DPSEL is $O(n\,2^n)$.

Subset enumeration (cf. lines 3 - 4 in Fig. 7) 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.

Map dependencies in System Tx are also stored in 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).

7. EXPERIMENTAL RESULTS

The evaluation of predicates in data warehouses has become the major bottleneck for decision support queries [14]. We show in this section that there is a huge optimization potential not harvested by other optimization algorithms. For the experimental evaluation of our optimization algorithm DPSel, we compared it against two widely used algorithms.

In some commercial systems, predicates are ordered simply by ascending selectivity. One example of such a system is Vectorwise [34, 37]. We term the algorithm that orders predicates in ascending order of their selectivity as Sel. Other systems order predicates in ascending order of their *rank* (cf. Eq. (1)). We call this algorithm Rank.

In this section, we are interested in answering three 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. (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 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 used inexpensive predicates with equal costs (special case). We enriched the experimental evaluation by running additional experiments using the TPC-H and the forest [5] 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 conjunctions containing up to 10 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 $[0, 1]$. Their consistency was ensured by means of PDHGMp [25]. 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) [22]. Appendix B shows how to compute predicate selectivities efficiently on fly, by means of sampling.

We conclude the Experiments section with a comparison of the running times of the three algorithms. The experiments were 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.

7.1 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 of 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 [1,1000].

We ran three different experiments, each time 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, a different predicates joint selectivity was picked from the pool of joint selectivities. For the first experiment, for each predicate we created a dependency graph containing a single subexpression. We assigned 1000 different random cost values to the subexpressions. We generated 100 different dependency graphs.

Since we were interested to find the maximum optimization potential of DPSel vs. the other two algorithms, we recorded the plans with the maximum cost difference from all the runs. We repeated the same experiment, where 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. As the plan costs varied greatly, the plan costs of Sel and Rank are given relative to DPSel. Thus, this table contains 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 such that for 10 predicates DPSel beats Sel by a whopping factor of 110, and Rank by a factor of 7.

7.2 Special case

In this section we list our experimental findings of comparing the qualities of plans generated by DPSel and the other two algorithms (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.

We have tested the algorithms starting with three and up to ten predicates, incrementally. 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 evaluation experiment, all the predicates were assigned dependency graphs containing a single subexpression. The results of this experimental evaluation are shown in Fig. 10. The y-axes show the per-tuple cost in nanoseconds (ns), whereas the x-axes show the number of predicates. The algorithms Sel and Rank produced the same results, due to the fact that 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 branch misprediction costs.

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 pred-

	Nr. of predicates															
	3		4		5		6		7		8		9		10	
Nr. subexpr.	Rank	Sel	Rank	Sel	Rank	Sel	Rank	Sel	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	5.1	38.4	5.8	64.3	6.4	80.5	6.9	110
2	2.6	2.6	3.1	3.9	3.5	8.3	4.2	15.9	5	21.3	5.7	29.6	6.3	35.2	7.2	42.7
3	2.6	2.6	3.1	3.1	3.4	3.4	3.8	3.8	4.3	4.3	4.8	4.8	5.3	5.3	5.7	5.7

Table 5: Relative optimization potential (in factors!) of DPSel vs. Rank and Sel

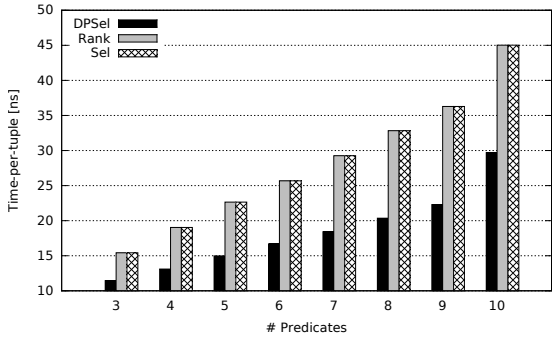

Figure 10: Plan costs for inexpensive predicates sharing a single subexpression

icates, the difference on plan qualities was as large as 40% in favor of DPSel. This is a large optimization potential, considering that we have evaluated inexpensive predicates.

We repeated the same experiment, but this time each selection depended on values of one unique subexpression. That is, there were no shared subexpressions among selections. This way, we have eliminated the optimization potential that DPSel harvests by employing CSE. The results of this experiment are shown in Fig. 11. We observe that the

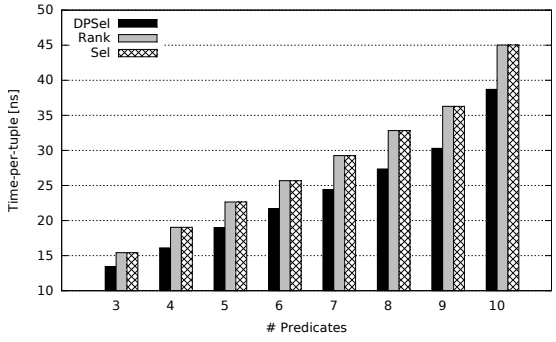

Figure 11: Plan costs for inexpensive predicates, no shared subexpression

costs of plans produced by DPSel are nevertheless lower than those of Rank and Sel. This time, though, DPSel produced better plans solely due to minimizing 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 Sec. 7.1, 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 Fig. 12. We observe similar results

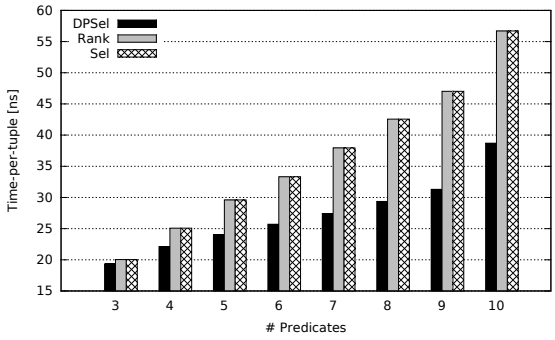

Figure 12: Plan costs for inexpensive predicates depending on 3 different subexpressions

to the case when a single subexpression was shared among all selections (see Fig. 10). 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.

7.2.1 TPC-H 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.

Algorithm	Est. plan cost [ns]	Evaluation cost [ns]
DPSel	8.49	8.99
Rank/Sel	12.09	12.59

Table 6: DPSel vs. Rank and Sel over TPC-H dataset

The results of this experiment are shown in Table 6. The second column of the table shows the estimated plan costs in time-per-tuple for each algorithm, whereas the third column shows the actual measured plan costs, by running the plans in System Tx.

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.

7.2.2 Forest dataset

In this section, we present the experimental evaluation of DPSel vs. Rank and Sel by using the forest [5] 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 random attributes of the forest relation, with random predicate constants (i.e., c_1, c_2). The results of this experiment are shown in Table 7. DPSel beats the other two heuristics algorithms

Algorithm	Equal costs	Varying costs
Rank/Sel	2.01	21.42

Table 7: Relative optimization potential of DPSel vs. Rank and Sel over the forest dataset

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, 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 7, third column).

7.3 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 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 the forest [5] 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 have been shown in Table 8. In the light of theorem 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 bellow q^2 for all values of f.

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

7.4 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 Fig. 13. The y-axis denotes the runtime in milliseconds (ms), where-

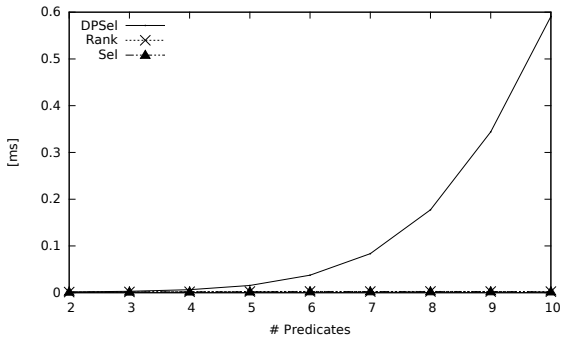

Figure 13: The evaluation results of runtime performance

as the x-axis denotes the number of predicates that were fed to the algorithms.

Although DPSel has $O(n\,2^n)$ complexity, its runtime for up to 10 predicates is 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 effort.

8. CONCLUSIONS AND FUTURE WORK

We presented the first 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 are the fundament of 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.

9. REFERENCES

[1] D. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization strategies in a

column-oriented DBMS. In *ICDE 2007*, pages 466–475, 2007.

[2] J. A. Blakeley, W. J. McKenna, and G. Graefe. Experiences building the open oodb query optimizer. In *SIGMOD*, volume 22, pages 287–296, 1993.

[3] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards estimation error guarantees for distinct values. In *PODS*, pages 268–279, 2000.

[4] S. Christodoulakis. Implications of certain assumptions in database performance evauation. *TODS*, pages 163–186, 1984.

[5] College of Natural Resources Colorado State University. Forest dataset. `http://kdd.ics.uci.edu/databases/covertype/covertype.data.html`.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, et al. *Introduction to algorithms*. MIT press Cambridge, 2001.

[7] G. Cormode, M. Garofalakis, P. Haas, and C. Jermaine. *Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches*. NOW Press, 2012.

[8] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD*, pages 1243–1254, 2013.

[9] P. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *VLDB*, pages 541–550, 2001.

[10] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. HYRISE: a main memory hybrid storage engine. *PVLDB*, pages 105–116, 2010.

[11] J. M. Hellerstein and M. Stonebraker. *Predicate migration: Optimizing queries with expensive predicates*, volume 22. ACM, 1993.

[12] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *TODS*, pages 482–502, 1984.

[13] IBM. Soliddb. `http://www.ibm.com/software/data/soliddb`.

[14] R. Johnson, V. Raman, R. Sidle, and G. Swart. Row-wise parallel predicate evaluation. *VLDB*, pages 622–634, 2008.

[15] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, pages 1496–1499, 2008.

[16] C.-C. Kanne and G. Moerkotte. Histograms reloaded: The merits of bucket diversity. In *SIGMOD*, pages 663–674, 2010.

[17] A. Kemper and G. Moerkotte. Advanced query processing in object bases using access support relations. In *VLDB*, pages 290–301, 1990.

[18] A. Kemper, G. Moerkotte, and M. Steinbrunn. Optimizing boolean expressions in object bases. In *VLDB*, pages 79–90, 1992.

[19] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[20] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *VLDB*, pages 128–137, 1986.

[21] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *VLDB*, pages 204–215, 2015.

[22] V. Markl, P. J. Haas, M. Kutsch, N. Megiddo, U. Srivastava, and T. M. Tran. Consistent selectivity estimation via maximum entropy. *The VLDB journal*, 16(1):55–76, 2007.

[23] V. Markl, G. Lohman, and V. Raman. LEO: An autonomic query optimizer for DB2. *IBM Systems Journal*, 42(1):98–106, 2003.

[24] G. Moerkotte. *Building Query Compiler*. 2014. pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf.

[25] G. Moerkotte, M. Montag, A. Repetti, and G. 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.

[26] G. Moerkotte, T. Neumann, and G. Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *VLDB*, pages 982–993, 2009.

[27] K. Munagala, S. Babu, R. Motwani, and J. Widom. The pipelined set cover problem. In *Database Theory-ICDT 2005*, pages 83–98. Springer, 2005.

[28] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, pages 539–550, 2011.

[29] T. Neumann, S. Helmer, and G. Moerkotte. On the optimal ordering of maps and selections under factorization. In *ICDE*, pages 490–501, 2005.

[30] Oracle. TimesTen In-Memory Database. `http://www.oracle.com/technetwork/database/database-technologies/timesten/overview/index.html`.

[31] K. A. Ross. Conjunctive selection conditions in main memory. In *SIGMOD*, pages 109–120, 2002.

[32] SAP. In-Memory Computing (SAP HANA). `http://www.sap.com/pc/tech/in-memory-computing-hana/software/overview/index.html`.

[33] S. Setzer, G. Steidl, T. Teuber, and G. Moerkotte. Approximation related to quotient functionals. *Journal of Approximation Theory*, pages 545–558, 2010.

[34] J. Sompolski, M. Zukowski, and P. Boncz. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, pages 33–40, 2011.

[35] K. Tzoumas, A. Deshpande, and C. Jensen. Efficiently adapting graphical models for selectivity estimation. *VLDB Journal*, 22:3–27, 2013.

[36] VoltDB. In-memory database. `http://www.voltdb.com`.

[37] M. Zukowski, M. Van de Wiel, and P. Boncz. Vectorwise: A vectorized analytical dbms. In *ICDE*, pages 1349–1350, 2012.

APPENDIX

A. System Tx

Although System Tx 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 [28]. 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++; ...
    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 bellow.

In System Tx, 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_MAP::step(t) {
  t.A = R.A[t.rid];
  t.B = R.B[t.rid];
  ...
  consumer.step(t);
}
```

The second method accesses column values based on column pointers

```
Tx_MAP::step(t) {
  t.A = *(t.ap);
  t.B = *(t.bp);
  ...
  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.

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

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

B. SAMPLING SELECTIVITIES

Our algorithm requires selectivity estimates for subsets of predicates. Selectivity estimates for subsets of predicates can be derived in several ways, e.g., by entropy maximization [22] or graphical models [35]. Both require some implementation effort and runtime. Next, we present a new, easy to implement, and very efficient alternative. The main idea is to extend the usual sampling procedure to gather more than the usual information.

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

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

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

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

The selectivities of these predicates are denoted by $s_{\beta(P')}$ and $s_{\gamma(P')}$. For our algorithm, we need the vector s_β, which gathers the $s_{\beta(P')}$ for all P'. The procedure `getGamma` presented below will give us s_γ. Hence, we need a method to convert s_γ to s_β.

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

Define the *complete design matrix* A (see also [22]) as

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

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 A is binary, non-singular, and persymmetric.

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

$$A s_\gamma = s_\beta.$$

Since the positions of the ones in row i can be enumerated efficiently by enumerating supersets of the bitvector i (see [24, p66] for details), multiplications of A with a vector x can be implemented very efficiently using only a few bit manipulating instructions and additions.

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. A simple piece of code (close to C++) shows how to do this:

```
getGamma(p, z, S)
  // p is vector of predicates,
  // z its length,
  // S is the sample
  int n = (1 << z);
  // array of counters initialized to zero
  int c_gamma[n] = {0};
  // for all sample tuples in S
  for(s : S)
    int k = 0;
    for(int i = 0; i < z; ++i)
      // p[i](s): evaluate pi on sample tuple s
      k |= (p[i](s) << i);
    ++c_gamma[k];
  return c_gamma/|S|; // componentwise division
```

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, k contains a bitpattern representing the outcome of all predicates. Then, k is used as an index into an array of counters and the according counter is increased. To get the selectivities s_γ, we must only divide these counters by the size of the sample.