VLDB Journal manuscript No.
(will be inserted by the editor)

Query Optimization Through the Looking Glass, and What We Found Running the Join Order Benchmark

Viktor Leis · Bernhard Radke · Andrey Gubichev · Atanas Mirchev · Peter Boncz · Alfons Kemper · Thomas Neumann

Received: date / Accepted: date

Abstract Finding a good join order is crucial for query performance. In this paper, we introduce the Join Order Benchmark (JOB) that works on real-life data riddled with correlations and introduces 113 complex join queries. We experimentally revisit the main components in the classic query optimizer architecture using a complex, real-world data set and realistic multi-join queries. For this purpose, we describe cardinality-estimate injection and extraction techniques that allow us to compare the cardinality estimators of multiple industrial SQL implementations on equal footing, and to characterize the value of having perfect cardinality estimates. Our investigation shows that all industrial-strength cardinality estimators routinely produce large errors: though cardinality estimation using table samples solves the problem for single-table queries, there are still no techniques in industrial systems that can deal accurately with join-crossing correlated query predicates. We further show that while estimates are essential for finding a good join order, query performance is unsatisfactory if the query engine relies too heavily on these estimates. Using another set of experiments that measure the impact of the cost model, we find that it has much less influence on query performance than the cardinality estimates. We investigate plan enumeration techniques comparing exhaustive dynamic programming with heuristic algorithms and find that exhaustive enumeration improves performance despite the sub-optimal cardinality estimates. Finally, we extend our investigation from main-memory only, to also include disk-based query processing. Here, we find that though accurate cardinality estimation should be the first priority, other aspects such as modeling random vs. sequential I/O are also important to predict query runtime.

1 Introduction

The problem of finding a good join order is one of the most studied problems in the database field. Fig. 1 illustrates the classical, cost-based approach, which dates back to System R [45]. To obtain an efficient query plan, the query optimizer enumerates some subset of the valid join orders, for example using dynamic programming. Using cardinality estimates as its principal input, the cost model then chooses the cheapest alternative from semantically equivalent plan alternatives.

Theoretically, as long as the cardinality estimations and the cost model are accurate, this architecture obtains the optimal query plan. In reality, cardinality estimates are usually computed based on simplifying assumptions like uniformity and independence. In real-world data sets, these assumptions are *frequently* wrong, which may lead to sub-optimal and sometimes disastrous plans.

Viktor Leis
Technische Universität München, Garching, Germany
E-mail: leis@in.tum.de

Bernhard Radke
Technische Universität München, Garching, Germany
E-mail: radke@in.tum.de

Andrey Gubichev
Technische Universität München, Garching, Germany
E-mail: gubichev@in.tum.de

Atanas Mirchev
Technische Universität München, Garching, Germany
E-mail: mirchev@in.tum.de

Peter Boncz
CWI, Amsterdam, The Netherlands
E-mail: p.boncz@cwi.nl

Alfons Kemper
Technische Universität München, Garching, Germany
E-mail: kemper@in.tum.de

Thomas Neumann
Technische Universität München, Garching, Germany
E-mail: neumann@in.tum.de

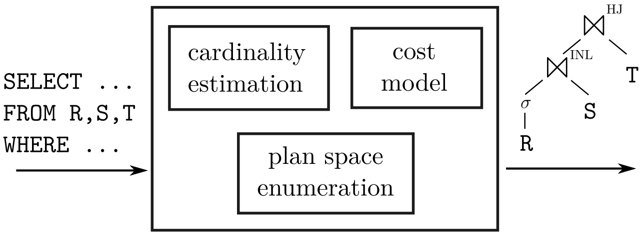


Fig. 1 Traditional query optimizer architecture.

In this paper we experimentally investigate the three main components of the classical query optimization architecture in order to answer the following questions:

- How good are cardinality estimators and when do bad estimates lead to slow queries?
- How important is an accurate cost model for the overall query optimization process?
- How large does the enumerated plan space need to be?

To answer these questions, we use a novel methodology that allows us to isolate the influence of the individual optimizer components on query performance. Our experiments are conducted using a real-world data set and 113 multi-join queries that provide a challenging, diverse, and realistic workload. The main contributions of this paper are:

- We design a challenging workload named *Join Order Benchmark (*JOB*)*, which is based on the IMDB data set. The benchmark is publicly available to facilitate further research.
- To the best of our knowledge, this work is the first end-to-end study of the join ordering problem using a real-world data set and realistic queries.
- By quantifying the contributions of cardinality estimation, the cost model, and the plan enumeration algorithm on query performance, we provide guidelines for the complete design of a query optimizer. We also show that many disastrous plans can easily be avoided.

The rest of this paper is organized as follows: We first discuss important background and the Join Order Benchmark in Section 2. Section 3 shows that the cardinality estimators of the major relational database systems produce bad estimates for many realistic queries, in particular for multi-join queries. The conditions under which these bad estimates cause slow performance are analyzed in Section 4. We show that it very much depends on how much the query engine relies on these estimates and on how complex the physical database design is, i.e., the number of indexes available. Query engines that mainly rely on hash joins and full table scans, are quite robust even in the presence of large cardinality estimation errors. The more indexes are available, the harder the problem becomes for the query optimizer resulting in runtimes that are far away from the optimal query plan. Section 5 shows that with the currently-used cardinality estimation techniques, the influence of cost model errors is dwarfed by cardinality estimation errors and that even quite simple cost models seem to be sufficient. Section 6 investigates different plan enumeration algorithms and shows that—despite large cardinality misestimates and sub-optimal cost models—exhaustive join order enumeration improves performance and that using heuristics leaves performance on the table. To augment the understanding obtained from aggregated statistics, Section 7 looks at two particular queries in our workload and analyzes their query plans. While most experiments use the in-memory setting, Section 8 repeats the important experiments with a cold cache and reading data from disk. Related work is discussed in Section 9.

We conclude this paper by repeating all important insights in Section 10. Time-constrained readers may start with Section 10, and selectively read the sections referenced there.

2 Background and Methodology

Many query optimization papers ignore cardinality estimation and only study search space exploration for join ordering with randomly generated, synthetic queries (e.g., [39, 14]). Other papers investigate only cardinality estimation in isolation either theoretically (e.g., [22]) or empirically (e.g., [53]). As important and interesting both approaches are for understanding query optimizers, they do not necessarily reflect real-world user experience.

The goal of this paper is to investigate the contribution of all relevant query optimizer components to end-to-end query performance in a realistic setting. We therefore perform our experiments using a workload based on a real-world data set and the widely-used PostgreSQL system. PostgreSQL is a relational database system with a fairly traditional architecture making it a good subject for our experiments. Furthermore, its open source nature allows one to inspect and change its internals. In this section we introduce the Join Order Benchmark, describe all relevant aspects of PostgreSQL, and present our methodology.

2.1 The IMDB Data Set

Many research papers on query processing and optimization use standard benchmarks like TPC-H, TPC-DS, or the Star Schema Benchmark (SSB) [4,43,41]. While these benchmarks have proven their value for evaluating query engines, we argue that they are not good benchmarks for the cardinality estimation component of query optimizers. The reason is that in order to easily be able to scale the benchmark data, the data generators are using the very same simplifying assumptions (uniformity, independence, principle of inclusion) that query optimizers make. Real-world data

Table 1 Cardinalities and aliases of the IMDB tables.

table	alias	cardinality
aka_name	an	901,343
aka_title	at	361,472
cast_info	ci	36,244,344
char_name	chn	3,140,339
comp_cast_type	cct	4
company_name	cn	234,997
company_type	ct	4
complete_cast	cc	135,086
info_type	it	113
keyword	k	134,170
kind_type	kt	7
link_type	lt	18
movie_companies	mc	2,609,129
movie_info	mi	14,835,720
movie_info_idx	mi_idx	1,380,035
movie_keyword	mk	4,523,930
movie_link	ml	29,997
name	n	4,167,491
person_info	pi	2,963,664
role_type	rt	12
title	t	2,528,312

sets, in contrast, are full of correlations and non-uniform data distributions, which makes cardinality estimation much harder. Section 3.3 shows that PostgreSQL's simple cardinality estimator indeed works unrealistically well for TPC-H. TPC-DS is slightly harder in that it has a number of non-uniformly distributed (skewed) attributes, but is still too easy due to not having correlations between attributes.

Therefore, instead of using a synthetic data set, we chose the *Internet Movie Data Base*[1] *(IMDB)*. It contains a plethora of information about movies and related facts about actors, directors, production companies, etc. The data is freely available[2] for non-commercial use as text files. In addition, we used the open-source *imdbpy*[3] package to transform the text files into a relational database. The schema and the key/foreign-key relationships are depicted in Fig. 2. The data set allows one to answer queries like "Which actors played in movies released between 2000 and 2005 with ratings above 8?". Like most real-world data sets IMDB is full of correlations and non-uniform data distributions, and is therefore much more challenging than most synthetic data sets. Our snapshot is from May 2013 and occupies 3.6 GB when exported to CSV files. The cardinalities of all 21 tables of the data set are listed in Table 1.

2.2 The JOB Queries

Based on the IMDB database, we have constructed analytical SQL queries. Each query consists of one select-project-join block[4]. Since we focus on join ordering, which arguably is the most important query optimization problem, we designed the queries to have between 3 and 16 joins, with an average of 8 joins per query. Query 13d, which is shown in Fig. 3, is a typical example that computes the ratings and release dates for all movies produced by US companies.

The join graph of query 13d is shown in Fig. 4. The solid edges in the graph represent key/foreign key edges ($1 : n$) with the arrow head pointing to the primary key side. Dotted edges represent foreign key/foreign key joins ($n : m$), which appear due to transitive join predicates. Our query set consists of 33 query structures, each with 2-6 variants that differ in their selections only, resulting in a total of 113 queries – all depicted in detail in Appendix A. Note that depending on the selectivities of the base table predicates, the variants of the same query structure have different optimal query plans that yield widely differing (sometimes by orders of magnitude) runtimes. Also, some queries have more complex selection predicates than the example (e.g., disjunctions or substring search using `LIKE`).

Our queries, which are shown in Appendix A, are "realistic" and "ad hoc" in the sense that they answer questions that may reasonably have been asked by a movie enthusiast. We also believe that despite their simple SPJ-structure, the queries model the core difficulty of the join ordering problem. For cardinality estimators the queries are challenging due to the significant number of joins and the correlations contained in the data set. However, we did not try to "trick" the query optimizer, e.g., by picking attributes with extreme correlations. Indeed we believe that real-world predicates on real-world data sets more often than not *are correlated* (i.e., not independent). The prevalence of correlated predicates is nicely illustrated by the well-known Honda Accord example [35], which is often used in IBM papers. We intentionally did not include more complex join predicates like inequalities or non-surrogate-key predicates, because cardinality estimation for JOB is already quite challenging.

We do not claim that the specific quantitative results obtained using JOB are directly transferable to other workloads. On the other hand, we also do not see any excuse for why the JOB should not run well in a relational database systems. Thus, we propose JOB for future research in cardinality estimation and join order optimization.

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

[2] `ftp://ftp.fu-berlin.de/pub/misc/movies/database/`

[3] `https://bitbucket.org/alberanid/imdbpy/get/5.0.zip`

[4] Since in this paper we do not model or investigate aggregation, we omitted `GROUP BY` from our queries. To avoid communication from becoming the performance bottleneck for queries with large result sizes, we wrap all attributes in the projection clause with `MIN(...)` expressions when executing (but not when estimating). This change has no effect on PostgreSQL's join order selection because its optimizer does not push down aggregations.

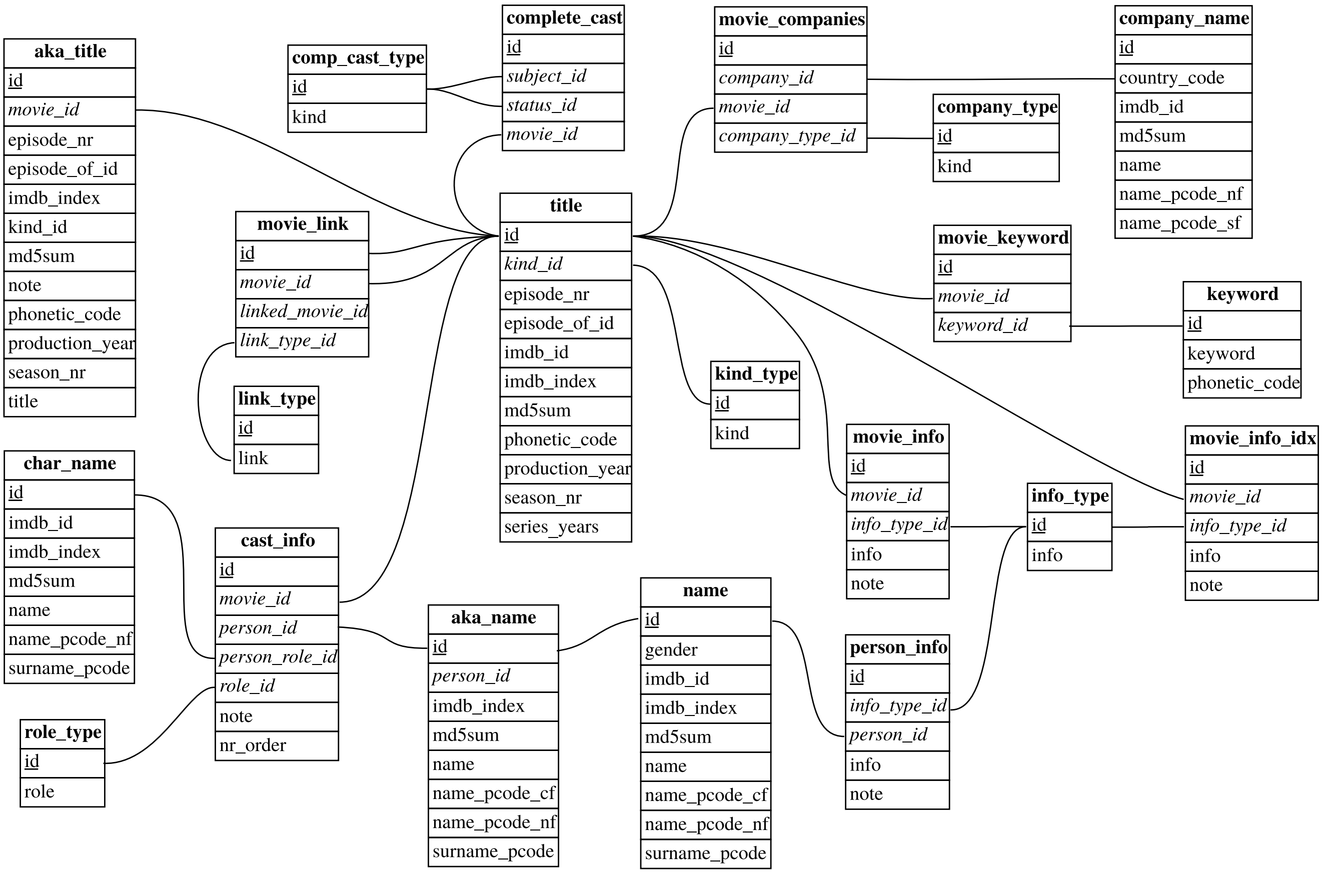


Fig. 2 IMDB schema with key/foreign-key relationships. Underlined attributes are primary keys. Italic font indicates a foreign-key attribute.

```
SELECT MIN(cn.name),
       MIN(mi.info),
       MIN(mi_idx.info)
FROM company_name cn, company_type ct,
     info_type it, info_type it2, title t,
     kind_type kt, movie_companies mc,
     movie_info mi, movie_info_idx mi_idx
WHERE cn.country_code = '[us]'
AND ct.kind = 'production companies'
AND it.info = 'rating'
AND it2.info = 'release dates'
AND kt.kind = 'movie'
AND .. --(11 join predicates/see Fig. 4)
```

Fig. 3 Example JOB query 13d computes the ratings and release dates for all movies produced by US companies.

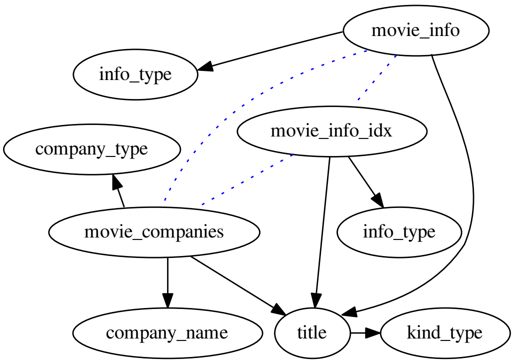


Fig. 4 Join graph for JOB queries 13a, 13b, 13c, 13d.

2.3 PostgreSQL

PostgreSQL's optimizer follows the traditional textbook architecture. Join orders, including bushy trees but excluding trees with cross products, are enumerated using dynamic programming. The cost model, which is used to decide which plan alternative is cheaper, is described in more detail in Section 5.1. The cardinalities of base tables are estimated using histograms (quantile statistics), most common values with their frequencies, and domain cardinalities (distinct value counts). These per-attribute statistics are computed by the `analyze` command using a sample of the relation. For complex predicates, where histograms can not be applied, the system resorts to ad hoc methods that are not theoretically grounded ("magic constants"). To combine conjunctive predicates for the same table, PostgreSQL simply assumes independence and multiplies the selectivities of the individual selectivity estimates.

The result sizes of joins are estimated using the formula

$$|T_1 \bowtie_{x=y} T_2| = \frac{|T_1||T_2|}{\max(\mathrm{dom}(x), \mathrm{dom}(y))},$$

where T_1 and T_2 are arbitrary expressions and $\mathrm{dom}(x)$ is the domain cardinality of attribute x, i.e., the number of distinct values of x. This value is the principal input for the join cardinality estimation. To summarize, PostgreSQL's cardinality estimator is based on the following assumptions:

- uniformity: all values, except for the most-frequent ones, are assumed to have the same number of tuples
- independence: predicates on attributes (in the same table or from joined tables) are independent
- principle of inclusion: the domains of the join keys overlap such that the keys from the smaller domain have matches in the larger domain

The query engine of PostgreSQL takes a physical operator plan and executes it using Volcano-style interpretation. The most important access paths are full table scans and lookups in unclustered B+Tree indexes. Joins can be executed using either nested-loop joins (with or without index lookups), in-memory hash joins, or sort-merge joins where the sort can spill to disk if necessary. The decision which join algorithm is used is made by the optimizer and cannot be changed at runtime.

2.4 Cardinality Extraction and Injection

We loaded the IMDB data set into 5 relational database systems: PostgreSQL, HyPer, and 3 commercial systems. Next, we ran the statistics gathering command of each database system with default settings to generate the database-specific statistics (e.g., histograms or samples) that are used by the estimation algorithms. We then obtained the cardinality estimates for all intermediate results of our test queries using database-specific commands (e.g., using the `EXPLAIN` command for PostgreSQL). We will later use these estimates of different systems to obtain optimal query plans (w.r.t. respective systems) and run these plans in PostgreSQL. For example, the intermediate results of the chain query

$$\sigma_{x=5}(A) \bowtie_{A.bid=B.id} B \bowtie_{B.cid=C.id} C$$

are $\sigma_{x=5}(A)$, $\sigma_{x=5}(A) \bowtie B$, $B \bowtie C$, and $\sigma_{x=5}(A) \bowtie B \bowtie C$. Additionally, the availability of indexes on foreign keys and index-nested-loop joins introduces the need for additional intermediate result sizes. For instance, if there exists a non-unique index on the foreign key $A.bid$, it is also necessary to estimate $A \bowtie B$ and $A \bowtie B \bowtie C$. The reason is that the selection $A.x = 5$ can only be applied *after* retrieving all matching tuples from the index on $A.bid$, and therefore the system produces two intermediate results, before and after the selection. Besides cardinality estimates from the different systems, we also obtain the true cardinality for each intermediate result by executing `SELECT COUNT(*)` queries[5].

We further modified PostgreSQL to enable cardinality injection of arbitrary join expressions, allowing its optimizer to use the estimates of other systems (or the true cardinalities) instead of its own. This allows one to directly measure the influence of cardinality estimates from different systems on query performance. Note that IBM DB2 supports a limited form of user control over the estimation process by allowing users to explicitly specify the selectivities of predicates. However, selectivity injection cannot fully model inter-relation correlations and is therefore less general than the capability of injecting cardinalities.

2.5 Experimental Setup

The cardinalities of the commercial systems were obtained using a laptop running Windows 7. All performance experiments were performed on a server with two Intel Xeon X5570 CPUs (2.9 GHz) and a total of 8 cores running PostgreSQL 9.4 on Linux. Version 9.4 does not support intra-query parallelism and since we do not execute multiple queries at the same time, only a single core was used in all experiments. The system has 64 GB of RAM, which means that the entire IMDB database is fully cached in RAM. Intermediate query processing results (e.g., hash tables) also easily fit into RAM, unless a very bad plan with extremely large intermediate results is chosen.

We set the memory limit per operator (`work_mem`) to 2 GB, which results in much better performance due to the more frequent use of in-memory hash joins instead of external memory sort-merge joins. Additionally, we set the buffer pool size (`shared_buffers`) to 4 GB and the size of the operating system's buffer cache used by PostgreSQL (`effective_cache_size`) to 32 GB. For PostgreSQL it is generally recommended to use OS buffering in addition to its own buffer pool and keep most of the memory on the OS side. The defaults for these three settings are very low (MBs, not GBs), which is why increasing them is generally recommended. Finally, by increasing the `geqo_threshold` parameter to 18 we forced PostgreSQL to always use dynamic programming instead of falling back to a heuristic for queries with more than 12 joins.

3 Cardinality Estimation

Cardinality estimates are the most important ingredient for finding a good query plan. Even exhaustive join order enu-

[5] For our workload it was still feasible to do this naïvely. For larger data sets the approach by Chaudhuri et al. [8] may become necessary.

Table 2 Q-errors for base table selections.

	median	90th	95th	max
PostgreSQL	1.00	2.08	6.10	207
DBMS A	1.01	1.33	1.98	43.4
DBMS B	1.00	6.03	30.2	104000
DBMS C	1.06	1677	5367	20471
HyPer	1.02	4.47	8.00	2084

meration and a perfectly accurate cost model are worthless unless the cardinality estimates are (roughly) correct. It is well known, however, that cardinality estimates are sometimes wrong by orders of magnitude, and that such errors are usually the reason for slow queries. In this section, we experimentally investigate the quality of cardinality estimates in relational database systems by comparing the estimates with the true cardinalities.

3.1 Estimates for Base Tables

To measure the quality of base table cardinality estimates, we use the *q-error*, which is the factor by which an estimate differs from the true cardinality. For example, if the true cardinality of an expression is 100, the estimates of 10 or 1000 both have a q-error of 10. Using the ratio instead of an absolute or quadratic difference captures the intuition that for making planning decisions only relative differences matter. The q-error furthermore provides a theoretical upper bound for the plan quality if the q-errors of a query are bounded [37].

Table 2 shows the 50th, 90th, 95th, and 100th percentiles of the q-errors for the 629 base table selections in our workload. The median q-error is close to the optimal value of 1 for all systems, indicating that the majority of all selections are estimated correctly. However, all systems produce misestimates for some queries, and the quality of the cardinality estimates differs strongly between the different systems.

Looking at the individual selections, we found that DBMS A and HyPer can usually predict even complex predicates like substring search using `LIKE` very well. To estimate the selectivities *for base tables* HyPer uses a random sample of 1000 rows per table and applies the predicates on that sample[6]. This allows one to get accurate estimates for arbitrary base table predicates as long as the selectivity is not too low. When we looked at the selections where DBMS A and HyPer produce errors above 2, we found that most of them have predicates with extremely low true selectivities (e.g., 10^{-5} or 10^{-6}). This routinely happens when the selection yields zero tuples on the sample, and the system falls back on an ad-hoc estimation method ("magic constants"). It therefore appears to be likely that DBMS A also uses the sampling approach.

The estimates of the other systems are worse and seem to be based on per-attribute histograms, which do not work well for many predicates and cannot detect (anti-)correlations between attributes. Note that we obtained all estimates using the default settings after running the respective statistics gathering tool. Some commercial systems support the use of sampling for base table estimation, multi-attribute histograms ("column group statistics"), or ex post feedback from previous query runs [47]. However, these features are either not enabled by default or are not fully automatic.

3.2 Estimates for Joins

Let us now turn our attention to the estimation of intermediate results for joins, which are more challenging because sampling or histograms do not work well. Fig. 5 summarizes over 100,000 cardinality estimates in a single figure. For each intermediate result of our query set, we compute the factor by which the estimate differs from the true cardinality, distinguishing between over- and underestimation. The graph shows one "boxplot" (note the legend in the bottom-left corner) for each intermediate result size, which allows one to compare how the errors change as the number of joins increases. The vertical axis uses a logarithmic scale to encompass underestimates by a factor of 10^8 and overestimates by a factor of 10^4.

Despite the better base table estimates of DBMS A, the overall variance of the join estimation errors, as indicated by the boxplot, is similar for all systems with the exception of DBMS B. For all systems we routinely observe misestimates by a factor of 1000 or more. Furthermore, as witnessed by the increasing height of the box plots, the errors grow exponentially (note the logarithmic scale) as the number of joins increases [22]. For PostgreSQL 16% of the estimates for 1 join are wrong by a factor of 10 or more. This percentage increases to 32% with 2 joins, and to 52% with 3 joins. For DBMS A, which has the best estimator of the systems we compared, the corresponding percentages are only marginally better at 15%, 25%, and 36%.

Another striking observation is that all tested systems—though DBMS A to a lesser degree—tend to systematically underestimate the results sizes of queries with multiple joins. This can be deduced from the median of the error distributions in Fig. 5. For our query set, it is indeed the case that the intermediate results tend to decrease with an increasing number of joins because more base table selections get applied. However, the true decrease is less than the independence assumption used by PostgreSQL (and apparently by the other systems) predicts. Underestimation is most pronounced with DBMS B, which frequently estimates 1 row for queries with more than 2 joins. The estimates of

[6] The sample is stored in the *DataBlock* [26] format, which enables fast scans (and therefore fast estimation). The sample is (re-)generated when computing the statistics of a table.

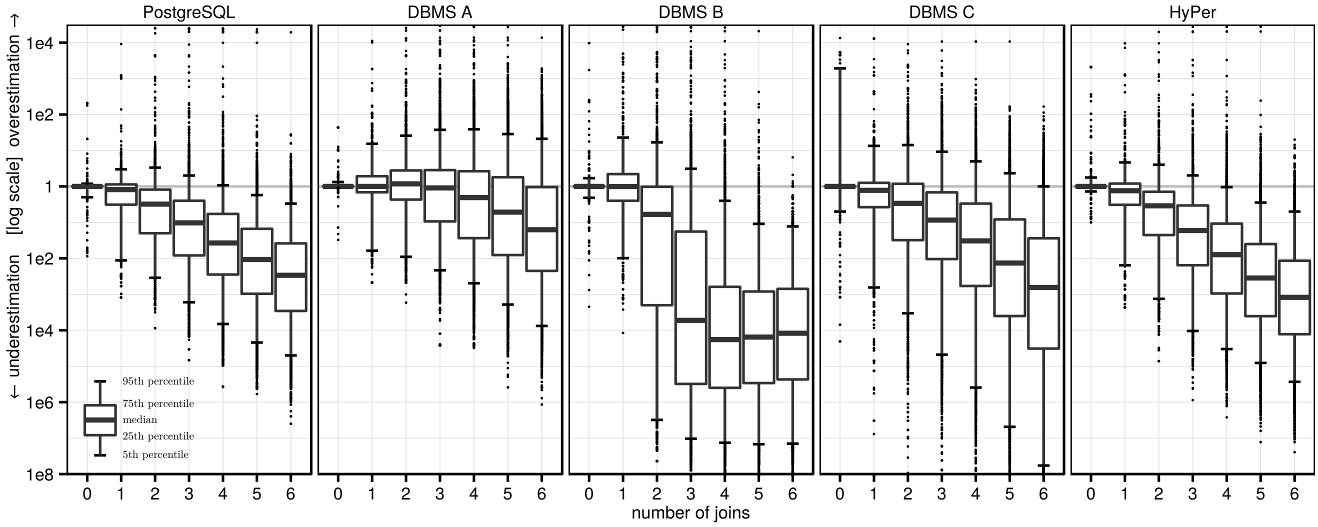


Fig. 5 Quality of cardinality estimates for multi-join queries in comparison with the true cardinalities. Each boxplot summarizes the error distribution of all subexpressions with a particular size (over all queries in the workload).

DBMS A, on the other hand, have medians that are much closer to the truth, despite their variance being similar to some of the other systems. We speculate that DBMS A uses a damping factor that depends on the join size, similar to how many optimizers combine multiple selectivities. Many estimators combine the selectivities of multiple predicates (e.g., for a base relation or for a subexpression with multiple joins) not by assuming full independence, but by adjusting the selectivities “upwards”, using a damping factor. The motivation for this stems from the fact that the more predicates need to be applied, the less certain one should be about their independence.

Given the simplicity of PostgreSQL’s join estimation formula (cf. Section 2.3) and the fact that its estimates are nevertheless competitive with the commercial systems, we can deduce that the current join size estimators are based on the independence assumption. No system tested was able to detect join-crossing correlations.

Note that this section does not benchmark the query optimizers of the different systems. In particular, our results do not imply that the DBMS B’s optimizer or the resulting query performance is necessarily worse than that of other systems, despite larger errors in the estimator. The query runtime heavily depends on how the system’s optimizer uses the estimates and how much trust it puts into these numbers. A sophisticated engine may employ adaptive operators (e.g., [5,9]) and thus mitigate the impact of misestimations, while another engine might have very complex access paths or join methods that require more accurate estimates. The results do, however, demonstrate that the state-of-the-art in cardinality estimation is far from perfect and its brittleness is further illustrated by the following anecdote: In PostgreSQL, we observed different cardinality estimates of the same simple 2-join query depending on the *syntactic* order of the relations in the `from` and/or the join predicates in the `where` clauses! Simply by swapping predicates or relations, we observed the estimates of 3, 9, 128, or 310 rows for the same query (with a true cardinality of 2600)[7].

3.3 Estimates for TPC-H

We have stated earlier that cardinality estimation in TPC-H is a rather trivial task. Fig. 6 substantiates that claim by showing the distributions of PostgreSQL estimation errors for 3 of the larger TPC-H queries and 4 of our JOB queries. Note that in the figure we report estimation errors for *individual* queries (not for all queries like in Fig. 5). Clearly, the TPC-H query workload does not present many hard challenges for cardinality estimators. In contrast, our workload contains queries that routinely lead to severe overestimation and underestimation errors, and hence can be considered a challenging benchmark for cardinality estimation.

3.4 Better Statistics for PostgreSQL

As mentioned in Section 2.3, the most important statistic for join estimation in PostgreSQL is the number of distinct values. These statistics are estimated from a fixed-sized sample, and we have observed severe underestimates for large tables. To determine if the misestimated distinct counts are

[7] The reasons for this surprising behavior are two implementation artifacts: First, estimates that are less than 1 are rounded up to 1, making subexpression estimates sensitive to the (usually arbitrary) join enumeration order, which is affected by the `from` clause. The second is a consistency problem caused by incorrect domain sizes of predicate attributes in joins with multiple predicates.

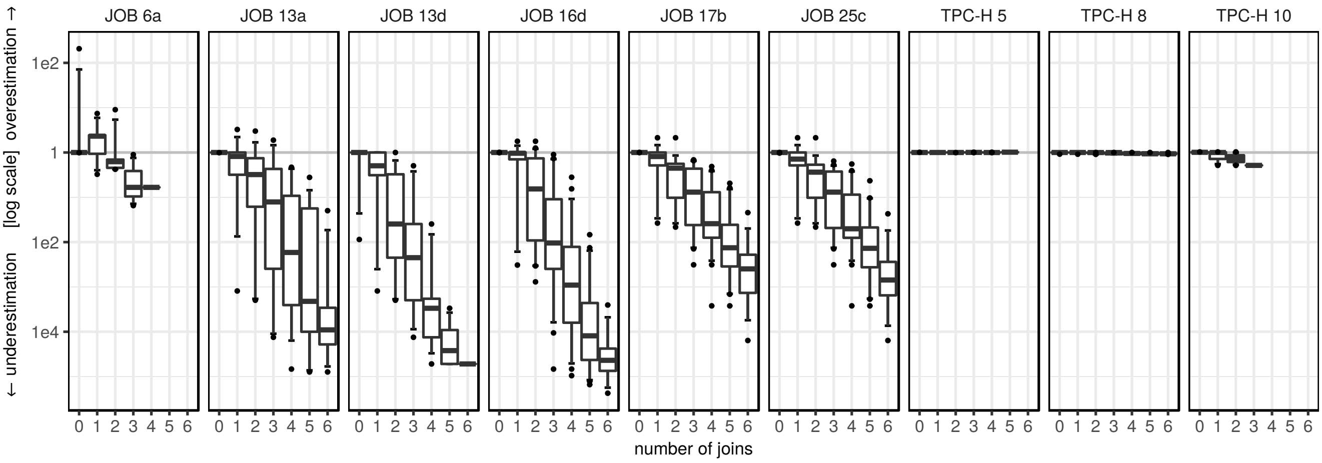


Fig. 6 PostgreSQL cardinality estimates for 6 JOB queries and 3 TPC-H queries.

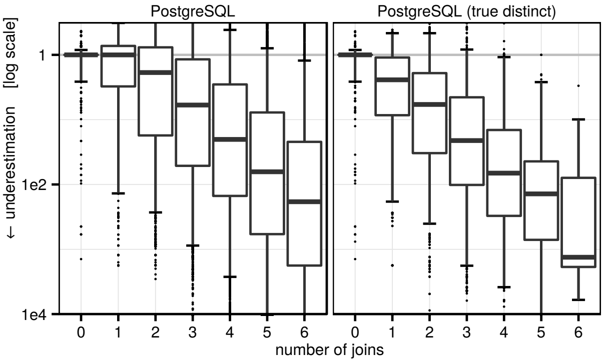


Fig. 7 PostgreSQL cardinality estimates based on the default distinct count estimates, and the true distinct counts.

the underlying problem for cardinality estimation, we computed these values precisely and replaced the estimated with the true values.

Fig. 7 shows that the true distinct counts slightly improve the variance of the errors. Surprisingly, however, the trend to underestimate cardinalities becomes even more pronounced. The reason is that the original, underestimated distinct counts resulted in higher estimates, which, accidentally, are closer to the truth. This is an example for the proverbial "two wrongs that make a right", i.e., two errors that (partially) cancel each other out. Such behavior makes analyzing and fixing query optimizer problems very frustrating because fixing one query might break another.

4 When Do Bad Cardinality Estimates Lead to Slow Queries?

While the large estimation errors shown in the previous section are certainly sobering, large errors do *not necessarily* lead to slow query plans. For example, the misestimated expression may be cheap in comparison with other parts of the query, or the relevant plan alternative may have been misestimated by a similar factor thus "canceling out" the original error. In this section we investigate the conditions under which bad cardinalities are likely to cause slow queries.

One important observation is that query optimization is closely intertwined with the physical database design: the type and number of indexes heavily influence the plan search space, and therefore affects how sensitive the system is to cardinality misestimates. We therefore start this section with experiments using a relatively robust physical design with only primary key indexes and show that in such a setup the impact of cardinality misestimates can largely be mitigated. After that, we demonstrate that for more complex configurations with many indexes, cardinality misestimation makes it much more likely to miss the optimal plan by a large margin.

4.1 The Risk of Relying on Estimates

To measure the impact of cardinality misestimation on query performance we injected the estimates of the different systems into PostgreSQL and then executed the resulting plans. Using the same query engine allows one to compare the cardinality estimation components in isolation by (largely) abstracting away from the different query execution engines. Additionally, we inject the true cardinalities, which computes the—with respect to the cost model—optimal plan. We group the runtimes based on their slowdown w.r.t. the optimal plan, and report the distribution in the following table, where each column corresponds to a group of slowdown factors (the group [2,10), for example, contains all queries where the slowdown is between 2 and 10):

	<0.9	[0.9,1.1)	[1.1,2)	[2,10)	[10,100)	>100
PostgreSQL	1.8%	38%	25%	25%	5.3%	5.3%
DBMS A	2.7%	54%	21%	14%	0.9%	7.1%
DBMS B	0.9%	35%	18%	15%	7.1%	25%
DBMS C	1.8%	38%	35%	13%	7.1%	5.3%
HyPer	2.7%	37%	27%	19%	8.0%	6.2%

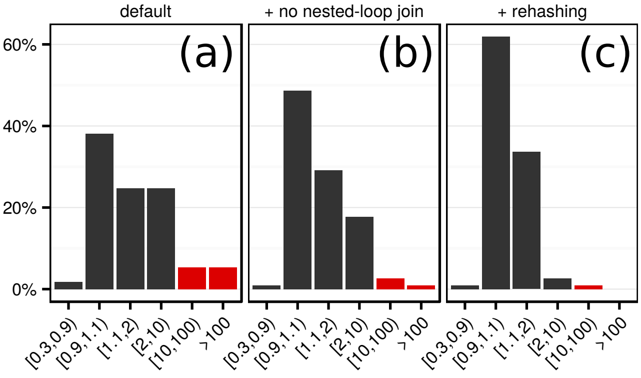


Fig. 8 Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities. Only primary key indexes are enabled and we modified the query engine: (a) shows the performance of PostgreSQL 9.4, (b) disables nested-loop joins that do not use indexes, and (c) additionally enables dynamic rehashing of the hash tables that are used in hash joins.

A small number of queries become slightly slower using the true instead of the erroneous cardinalities. This effect is caused by cost model errors, which we discuss in Section 5. However, as expected, the vast majority of the queries are slower when estimates are used. Using DBMS A's estimates, 78% of the queries are less than 2× slower than using the true cardinalities, while for DBMS B this is the case for only 53% of the queries. This corroborates the findings about the relative quality of cardinality estimates in the previous section. Unfortunately, all estimators occasionally lead to plans that take an unreasonable time and lead to a timeout. Surprisingly, however, many of the observed slowdowns are easily avoidable despite the bad estimates as we show in the following.

When looking at the queries that did not finish in a reasonable time using the estimates, we found that most have one thing in common: PostgreSQL's optimizer decides to introduce a nested-loop join (without an index lookup) because of a very low cardinality estimate, whereas in reality the true cardinality is larger. As we saw in the previous section, systematic underestimation happens very frequently, which occasionally results in the introduction of nested-loop joins.

The underlying reason why PostgreSQL chooses nested-loop joins is that it picks the join algorithm on a purely cost-based basis. For example, if the cost estimate is 1,000,000 with the nested-loop join algorithm and 1,000,001 with a hash join, PostgreSQL will always prefer the nested-loop algorithm even if there is a equality join predicate, which allows one to use hashing. Of course, given the $O(n^2)$ complexity of nested-loop join and $O(n)$ complexity of hash join, and given the fact that underestimates are quite frequent, this decision is extremely risky. And even if the estimates happen to be correct, any potential performance advantage of a nested-loop join in comparison with a hash join is very small, so taking this *high risk* can only result in a *very small payoff*.

Therefore, we disabled nested-loop joins (but not index-nested-loop joins) in all following experiments. As Fig. 8b shows, when rerunning all queries without these risky nested-loop joins, we observed no more timeouts despite using PostgreSQL's estimates.

Also, none of the queries performed slower than before despite having less join algorithm options, confirming our hypothesis that nested-loop joins (without indexes) seldom have any upside. However, this change does not solve all problems, as there are still a number of queries that are more than a factor of 10 slower (cf., red bars) in comparison with the true cardinalities.

When investigating the reason why the remaining queries still did not perform as well as they could, we found that most of them contain a hash join where the size of the build input is underestimated. PostgreSQL up to and including version 9.4 chooses the size of the in-memory hash table based on the cardinality estimate. Underestimates can lead to undersized hash tables with very long collisions chains and therefore bad performance. The upcoming version 9.5 resizes the hash table at runtime based on the number of rows actually stored in the hash table. We backported this patch to our code base, which is based on 9.4, and enabled it for all remaining experiments. Fig. 8c shows the effect of this change in addition with disabled nested-loop joins. Less than 4% of the queries are off by more than 2× in comparison with the true cardinalities.

To summarize, being "purely cost-based", i.e., not taking into account the inherent uncertainty of cardinality estimates and the asymptotic complexities of different algorithm choices, can lead to very bad query plans. Algorithms that seldom offer a large benefit over more robust algorithms should not be chosen. Furthermore, query processing algorithms should, if possible, automatically determine their parameters at runtime instead of relying on cardinality estimates.

4.2 Good Plans Despite Bad Cardinalities

The query runtimes of plans with different join orders often vary by many orders of magnitude (cf. Section 6.1). Nevertheless, when the database has only primary key indexes, as in all experiments so far, and once nested-loop joins have been disabled and rehashing has been enabled, the perfor-

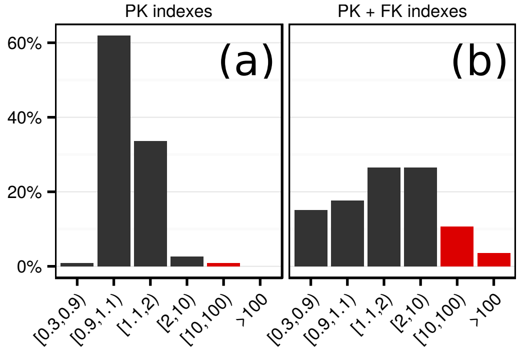


Fig. 9 Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities. Nested-loop joins that do not use indexes are disabled and rehashing is enabled, i.e., plot (a) is the same as Fig. 8(c). Adding foreign key indexes makes finding optimal plans much more difficult.

mance of most queries is close to the one obtained using the true cardinalities. Given the bad quality of the cardinality estimates, we consider this to be a surprisingly positive result. It is worthwhile to reflect on why this is the case.

The main reason is that without foreign key indexes, most large ("fact") tables need to be scanned using full table scans, which dampens the effect of different join orders. The join order still matters, but the results indicate that the cardinality estimates are usually good enough to rule out all disastrous join order decisions like joining two large tables using an unselective join predicate. Another important reason is that in main memory picking an index-nested-loop join where a hash join would have been faster is never disastrous. With all data and indexes fully cached, we measured that the performance advantage of a hash join over an index-nested-loop join is at most 5× with PostgreSQL and 2× with HyPer. Obviously, when the index must be read from disk, random IO may result in a much larger factor. Therefore, the main-memory setting is much more forgiving.

4.3 Complex Access Paths

So far, all query executions were performed on a database with indexes on primary key attributes only. To see if the query optimization problem becomes harder when there are more indexes, we additionally indexed all foreign key attributes. Fig. 9b shows the effect of additional foreign key indexes. We see large performance differences with 40% of the queries being slower by a factor of 2! Note that these results do not mean that adding more indexes decreases performance (although this can occasionally happen). Indeed overall performance generally increases significantly, but the more indexes are available the harder the job of the query optimizer becomes.

4.4 Join-Crossing Correlations

There is consensus in our community that estimation of intermediate result cardinalities in the presence of *correlated* query predicates is a frontier in query optimization research. Many industrial query optimizers keep only individual column statistics (e.g., histograms) and use the independence assumption for combining predicates on multiple columns. There has been previous work in detecting correlations between value distributions of different columns in the same table, for which then multi-column histograms or samples can be kept, e.g. [20]. The JOB workload studied in this paper consists of real-world data and its queries contain many correlated predicates. Our experiments that focus on *single-table* subquery cardinality estimation quality (cf. Table 2) show that systems that keep table samples (HyPer and presumably DBMS A) can achieve almost perfect estimation results, even for correlated predicates (inside the same table). As such, the cardinality estimation research challenge appears to lie in queries where the correlated predicates involve columns from *different* tables, connected by joins. These we call "join-crossing correlations". Such correlations frequently occur in the IMDB data set, e.g., actors born in Paris are likely to play in French movies.

Given these join-crossing correlations one could wonder if there exist complex access paths that allow one to exploit these. One example relevant here despite its original setting in XQuery processing is ROX [23]. It studied runtime join order query optimization in the context of DBLP co-authorship queries that count how many `Authors` had published `Papers` in three particular venues, out of many. These queries joining the author sets from different venues clearly have join-crossing correlations, since authors who publish in `VLDB` are typically database researchers, likely to also publish in `SIGMOD`, but not—say—in `Nature`.

In the DBLP case, `Authorship` is a $n : m$ relationship that links the relation `Authors` with the relation `Papers`. The optimal query plans in [23] used an index-nested-loop join, looking up each `author` into `Authorship.author` (the leading column of the indexed primary key of `Authorship`) followed by a filter restriction on `Paper.venue`, which needs to be looked up with yet another join. This filter on venue would normally have to be calculated *after* these two joins. However, the physical design of [23] stored `Authorship` *partitioned by* `Paper.venue`.[8] This partitioning has startling effects: instead of one `Authorship` table and primary key index, one physically has many, one for each `venue` partition. This means that by accessing the right

[8] In fact, rather than relational table partitioning, there was a separate XML document per venue, e.g., separate documents for `SIGMOD`, `VLDB`, `Nature` and a few thousand more venues. Storage in a separate XML document has roughly the same effect on access paths as partitioned tables.

partition, the filter is implicitly enforced (for free), *before* the join happens. This specific physical design therefore causes the optimal plan to be as follows: first join the smallish authorship set from `SIGMOD` with the large set for `Nature` producing almost no result tuples, making the subsequent nested-loops index lookup join into `VLDB` very cheap. If the tables would not have been partitioned, index lookups from all `SIGMOD` authors into `Authorships` would first find *all* co-authored papers, of which the great majority is irrelevant because they are from database venues, and were not published in `Nature`. Without this partitioning, there is no way to avoid this large intermediate result, and there is no query plan that comes close to the partitioned case in efficiency: even if cardinality estimation would be able to predict join-crossing correlations, there would be no physical way to profit from this knowledge.

The lesson to draw from this example is that the effects of query optimization are always gated by the available options in terms of access paths. This is similar to our experiments with and without indexes on foreign keys, where the latter, richer, scenario is more challenging for optimizers. Having a partitioned index on a *join-crossing correlated predicate* as in [23] is a non-obvious physical design alternative which even modifies the schema by bringing in a join-crossing column (`Paper.venue`) as partitioning key of a table (`Authorship`). We did not try to apply such optimizations in our IMDB experiments, because a phyiscal design similar to [23] would help only a minority of the join-crossing correlations in our 113 queries, and this type of indexing is by no means common practice. The partitioned DBLP set-up is just one example of how one particular join-crossing correlation can be handled, rather than a generic solution. Join-crossing correlations remain an open frontier for database research involving the interplay of physical design, query execution and query optimization. In our JOB experiments we do not attempt to chart this mostly unknown space, but rather characterize the impact of (join-crossing) correlations on the current state-of-the-art of query processing, restricting ourselves to standard PK and FK indexing.

5 Cost Models

The cost model guides the selection of plans from the search space. The cost models of contemporary systems are sophisticated software artifacts that are resulting from 30+ years of research and development, mostly concentrated in the area of traditional disk-based systems. PostgreSQL's cost model, for instance, is comprised of over 4000 lines of C code, and takes into account various subtle considerations, e.g., it takes into account partially correlated index accesses, interesting orders, tuple sizes, etc. It is interesting, therefore, to evaluate how much a complex cost model actually contributes to the overall query performance.

First, we will experimentally establish the correlation between the PostgreSQL cost model—a typical cost model of a disk-based DBMS—and the query runtime. Then, we will compare the PostgreSQL cost model with two other cost functions. The first cost model is a tuned version of PostgreSQL's model for a main-memory setup where all data fits into RAM. The second cost model is an extremely simple function that only takes the number of tuples produced during query evaluation into account. We show that, unsurprisingly, the difference between the cost models is dwarfed by the cardinality estimates errors. We conduct our experiments on a database instance with foreign key indexes. We begin with a brief description of a typical disk-oriented complex cost model, namely the one of PostgreSQL.

5.1 The PostgreSQL Cost Model

PostgreSQL's disk-oriented cost model combines CPU and I/O costs with certain weights. Specifically, the cost of an operator is defined as a weighted sum of the number of accessed disk pages (both sequential and random) and the amount of data processed in memory. The cost of a query plan is then the sum of the costs of all operators. The default values of the weight parameters used in the sum (*cost variables*) are set by the optimizer designers and are meant to reflect the relative difference between random access, sequential access and CPU costs.

The PostgreSQL documentation contains the following note on cost variables: *"Unfortunately, there is no well-defined method for determining ideal values for the cost variables. They are best treated as averages over the entire mix of queries that a particular installation will receive. This means that changing them on the basis of just a few experiments is very risky."* For a database administrator, who needs to actually set these parameters these suggestions are not very helpful; no doubt most will not change these parameters. This comment is of course, not PostgreSQL-specific, since other systems feature similarly complex cost models. In general, tuning and calibrating cost models (based on sampling, various machine learning techniques etc.) has been a subject of a number of papers (e.g, [51,32]). It is important, therefore, to investigate the impact of the cost model on the overall query engine performance. This will indirectly show the contribution of cost model errors on query performance.

5.2 Cost and Runtime

The main virtue of a cost function is its ability to predict which of the alternative query plans will be the fastest, given the cardinality estimates; in other words, what counts is its correlation with the query runtime. The correlation between the cost and the runtime of queries in PostgreSQL is shown

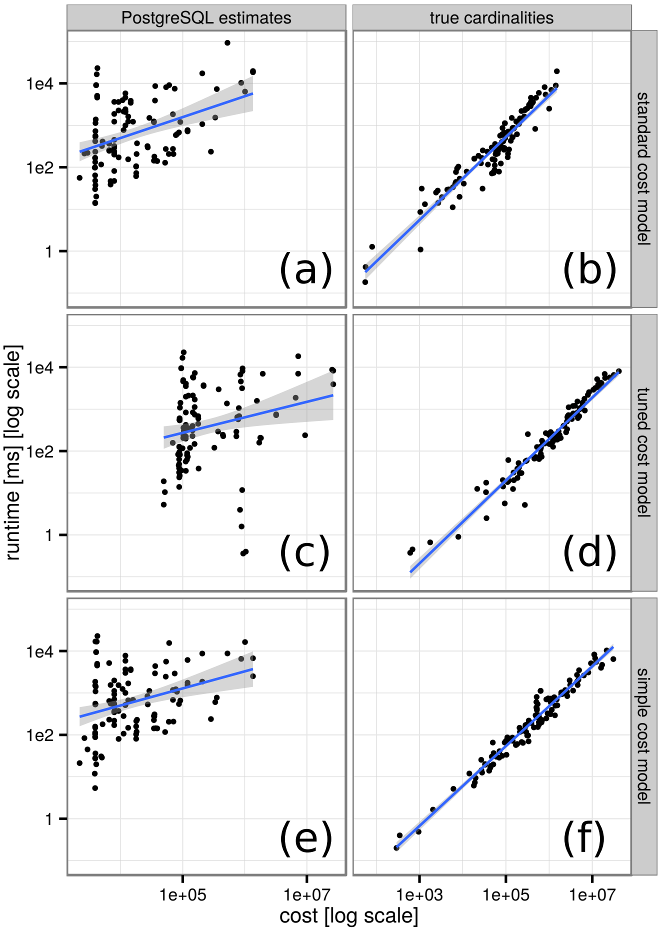


Fig. 10 Predicted cost vs. runtime for different cost models. Cardinality estimation has a much larger effect on query performance than the cost model and even a simple cost model seems sufficient for in-memory workloads.

in Fig. 10a. Additionally, we consider the case where the engine has the true cardinalities injected, and plot the corresponding data points in Fig. 10b. For both plots, we fit the linear regression model (displayed as a straight line) and highlight the standard error. The predicted cost of a query correlates with its runtime in both scenarios. Poor cardinality estimates, however, lead to a large number of outliers and a very wide standard error area in Fig. 10a. Only using the true cardinalities makes the PostgreSQL cost model a reliable predictor of the runtime, as has been observed previously [51].

Intuitively, a straight line in Fig. 10 corresponds to an ideal cost model that always assigns (predicts) higher costs for more expensive queries. Naturally, any monotonically increasing function would satisfy that requirement, but the linear model provides the simplest and the closest fit to the observed data. We can therefore interpret the deviation from this line as the *prediction error* of the cost model. Specifically, we consider the absolute percentage error of a cost model for a query Q: $\epsilon(Q) = \frac{|T_{\text{real}}(Q) - T_{\text{pred}}(Q)|}{T_{\text{real}}(Q)}$, where T_{real} is the observed runtime, and T_{pred} is the runtime predicted by our linear model. Using the default cost model of PostgreSQL and the true cardinalities, the median error of the cost model is 38%.

5.3 Tuning the Cost Model for Main Memory

As mentioned above, a cost model typically involves parameters that are subject to tuning by the database administrator. In a disk-based system such as PostgreSQL, these parameters can be grouped into *CPU cost parameters* and *I/O cost parameters*, with the default settings reflecting an expected proportion between these two classes in a hypothetical workload.

In many settings the default values are sub-optimal. For example, the default parameter values in PostgreSQL suggest that processing a tuple is 400x cheaper than reading it from a page. However, modern servers are frequently equipped with very large RAM capacities, and in many workloads the data set actually fits entirely into available memory (admittedly, the core of PostgreSQL was shaped decades ago when database servers only had a few megabytes of RAM). This does not eliminate the page access costs entirely (due to buffer manager overhead), but significantly bridges the gap between the I/O and CPU processing costs.

Arguably, the most important change that needs to be done in the cost model for a main-memory workload is to decrease the proportion between these two groups. We have done so by multiplying the *CPU cost* parameters by a factor of 50[9]. The results of the workload run with improved parameters are plotted in the two middle subfigures of Fig. 10. Comparing Fig. 10b with d, we see that tuning does indeed improve the correlation between the cost and the runtime. On the other hand, as is evident from comparing Fig. 10c and d, parameter tuning improvement is still overshadowed by the difference between the estimated and the true cardinalities. Note that Fig. 10c features a set of outliers for which the optimizer has accidentally discovered very good plans (runtimes around 1 ms) without realizing it (hence very high costs). This is another sign of “oscillation” in query planning caused by cardinality misestimates.

In addition, we measure the prediction error ϵ of the tuned cost model, as defined in Section 5.2. We observe that tuning improves the predictive power of the cost model: the median error decreases from 38% to 30%.

[9] We did not run extensive experiments to find the “best” parameter and do not claim that 50 is the best setting. Our goal is to measure the effect of adjusting cost parameters into a significantly more accurate direction to determine how important it is to tune the cost model.

5.4 Are Complex Cost Models Necessary?

As discussed above, the PostgreSQL cost model is quite complex. Presumably, this complexity should reflect various factors influencing query execution, such as the speed of a disk seek and read, CPU processing costs, etc. In order to find out whether this complexity is actually necessary in a main-memory setting, we will contrast it with a very simple cost function C_{mm}. This cost function is tailored for the *main-memory* setting in that it does not model I/O costs, but only counts the number of tuples that pass through each operator during query execution:

$$C_{\mathrm{mm}}(T) = \begin{cases} \tau \cdot |R| & \text{if } T = R \vee T = \sigma(R) \\ |T| + |T_1| + C_{\mathrm{mm}}(T_1) + C_{\mathrm{mm}}(T_2) & \text{if } T = T_1 \bowtie^{\mathrm{HJ}} T_2 \\ C_{\mathrm{mm}}(T_1) + & \text{if } T = T_1 \bowtie^{\mathrm{INL}} T_2, \\ \qquad \lambda \cdot |T_1| \cdot \max(\frac{|T_1 \bowtie R|}{|T_1|}, 1) & (T_2 = R \vee T_2 = \sigma(R)) \end{cases}$$

In the formula above R is a base relation, and $\tau \leq 1$ is a parameter that discounts the cost of a table scan in comparison with joins. The cost function distinguishes between hash $\bowtie^{\mathrm{HJ}}$ and index-nested-loop $\bowtie^{\mathrm{INL}}$ joins: the latter scans T_1 and performs index lookups into an index on R, thus avoiding a full table scan of R. A special case occurs when there is a selection on the right side of the index-nested-loop join, in which case we take into account the number of tuple lookups in the base table index and essentially discard the selection from the cost computation (hence the multiplier $\max(\frac{|T_1 \bowtie R|}{|T_1|}, 1)$). For index-nested-loop joins we use the constant $\lambda \geq 1$ to approximate by how much an index lookup is more expensive than a hash table lookup. Specifically, we set $\lambda = 2$ and $\tau = 0.2$. As in our previous experiments, we disable nested-loop joins when the inner relation is not an index lookup.

The results of our workload run with C_{mm} as a cost function are depicted in Fig. 10e and f. We see that even our trivial cost model is able to fairly accurately predict the query runtime using the true cardinalities. To quantify this argument, we measure the improvement in the runtime achieved by changing the cost model for true cardinalities: In terms of the geometric mean over all queries, our tuned cost model yields 41% faster runtimes than the standard PostgreSQL model, but even a simple C_{mm} makes queries 34% faster than the built-in cost function. This improvement is not insignificant, but on the other hand, it is dwarfed by improvement in query runtime observed when we replace estimated cardinalities with the real ones (cf. Fig. 8b). This allows us to reiterate our main message that cardinality estimation is much more crucial than the cost model.

6 Plan Space

Besides cardinality estimation and the cost model, the final important query optimization component is a plan enumeration algorithm that explores the space of semantically equivalent join orders. Many different algorithms, both exhaustive (e.g., [36,13]) as well as heuristic (e.g, [46,39]) have been proposed. These algorithms consider a different number of candidate solutions (that constitute the *search space*) when picking the best plan. In this section we investigate how large the search space needs to be in order to find a good plan.

The experiments of this section use a standalone query optimizer, which implements Dynamic Programming (DP) and a number of heuristic join enumeration algorithms. Our optimizer allows the injection of arbitrary cardinality estimates. In order to fully explore the search space, we do not actually execute the query plans produced by the optimizer in this section, as that would be infeasible due to the number of joins our queries have. Instead, we first run the query optimizer using the estimates as input. Then, we recompute the cost of the resulting plan with the true cardinalities, giving us a very good approximation of the runtime the plan would have in reality.

We use the in-memory cost model from Section 5.4 and assume that it perfectly predicts the query runtime, which, for our purposes, is a reasonable assumption since the errors of the cost model are negligible in comparison to the cardinality errors. This approach allows us to compare a large number of plans without executing all of them.

Note that due to a data handling mistake, the numbers reported in Section 7 of the conference version of this paper [28] differ from the ones reported in this section. However, the qualitative conclusions are largely unaffected.

6.1 How Important Is the Join Order?

We use the Quickpick [49] algorithm to visualize the costs of different join orders. Quickpick is a simple, randomized algorithm that picks join edges at random until all joined relations are fully connected. Each run produces a correct, but usually slow, query plan. By running the algorithm 10,000 times per query and computing the costs of the resulting plans, we obtain an approximate distribution for the costs of random plans [49]. Fig. 11 shows density plots for 6 representative example queries and for three physical database designs: no indexes, primary key indexes only, and primary+foreign key indexes. The costs are normalized by the optimal plan (with foreign key indexes), which we obtained by running dynamic programming and the true cardinalities.

The graphs, which use a logarithmic scale on the horizontal cost axis, clearly illustrate the importance of the join

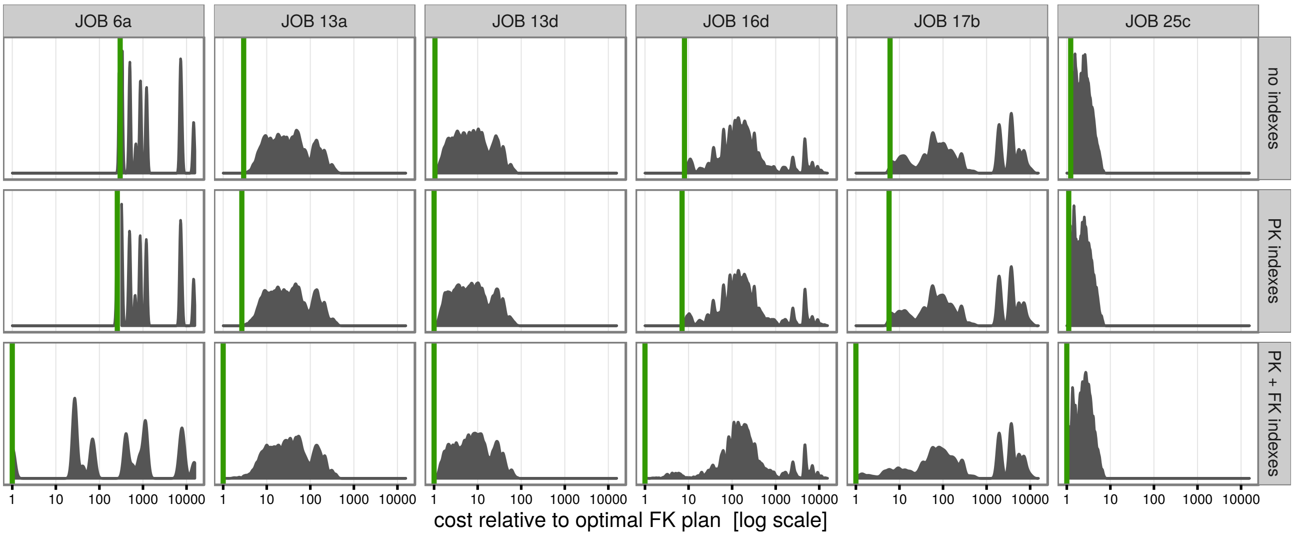


Fig. 11 Cost distributions for 6 queries and different index configurations. The vertical green lines represent the cost of the optimal plan.

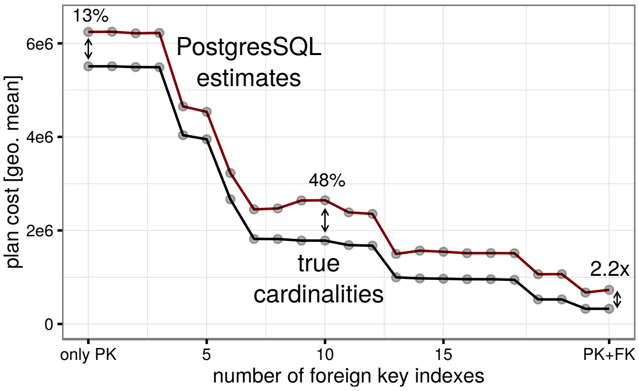


Fig. 12 Geometric mean of plan cost with varying index configurations.

ordering problem: The slowest or even median cost is generally multiple orders of magnitude more expensive than the cheapest plan. The shapes of the distributions are quite diverse. For some queries, there are many good plans (e.g., 25c), for others few (e.g., 16d). The distributions are sometimes wide (e.g., 16d) and sometimes narrow (e.g., 25c). The plots for the "no indexes" and the "PK indexes" configurations are very similar implying that for our workload primary key indexes alone do not improve performance very much, since we do not have selections on primary key columns. In many cases the "PK+FK indexes" distributions have additional small peaks on the left side of the plot, which means that the optimal plan in this index configuration is much faster than in the other configurations.

Indexes only on primary keys and on foreign as well as primary keys are two extremes of the spectrum. In reality, one would typically have only some foreign keys indexes. We therefore generated a random permutation of all foreign key indexes and enabled them one-by-one. The effect on plan quality as measured by the geometric mean of the plan cost is shown in Fig. 12. Generally, the more indexes exist, the better performance is. However, with PostgreSQL adding an index (e.g., 9) may even cause average performance to decrease (slightly). The curve is not smooth but has a number of large "jumps" due to the fact that certain foreign key indexes are particularly important for query performance. One final important point is that the relative performance difference increases dramatically from 13% to 2.2× as indexes are added because having more indexes makes finding the optimal plan harder.

6.2 Are Bushy Trees Necessary?

Most join ordering algorithms do not enumerate all possible tree shapes. Virtually all optimizers ignore join orders with cross products, which results in a dramatically reduced optimization time with only negligible query performance impact. Oracle goes even further by not considering bushy join trees [1], and System R only enumerated pipelined (right-deep) query plans. We define the left input of a join to be the build side of the hash join and the right input to be the probe side. In order to quantify the effect of restricting the search space on query performance, we modified our DP algorithm to only enumerate *left-deep*, *right-deep*, or *zig-zag* trees.

Aside from the obvious tree shape restriction, each of these classes implies constraints on the join method selection. We follow the definition of Garcia-Molina et al.'s textbook, which is reverse from the one in Ramakrishnan and Gehrke's book: Using hash joins, right-deep trees are executed by first creating hash tables out of each relation except one before probing in all of these hash tables in a pipelined fashion, whereas in left-deep trees, a new hash table is built from the result of each join. In zig-zag trees, which are a

Table 3 Slowdown for restricted tree shapes in comparison to the optimal plan (true cardinalities).

	PK indexes			PK + FK indexes		
	median	95%	max	median	95%	max
zig-zag	1.00	1.04	1.38	1.00	1.16	1.88
left-deep	1.00	1.23	1.66	1.94	48.2	1252
right-deep	1.12	1.63	1.69	6.46	140	6108

super set of all left- and right-deep trees, each join operator must have at least one base relation as input. For index-nested-loop joins we additionally employ the following convention: the left child of a join is a source of tuples that are looked up in the index on the right child, which must be a base table.

Using the true cardinalities, we compute the cost of the optimal plan for each of the three restricted tree shapes. We divide these costs by the optimal tree (which may have any shape, including "bushy") thereby measuring how much performance is lost by restricting the search space. The results in Table 3 show that zig-zag trees offer decent performance in most cases, with the worst case being 1.88× more expensive than the best bushy plan. Left-deep trees are worse than zig-zag trees, as expected, but still result in reasonable performance. Right-deep trees, on the other hand, perform much worse than the other tree shapes and thus should not be used exclusively. The bad performance of right-deep trees is caused by the large intermediate hash tables that need to be created from each base relation and the fact that only the bottom-most join can be done via index lookup.

6.3 Are Heuristics Good Enough?

So far in this paper, we have used the dynamic programming algorithm, which computes the optimal join order. However, given the bad quality of the cardinality estimates, one may reasonably ask whether an exhaustive algorithm is even necessary. We therefore compare dynamic programming with a randomized approach and two greedy heuristics.

The "Quickpick-1000" heuristics is a randomized algorithm that chooses the cheapest (based on the estimated cardinalities) 1000 random plans. Among all greedy heuristics, we pick Greedy Operator Ordering (GOO) since it was shown to be superior to other deterministic approximate algorithms [12]. GOO maintains a set of join trees, each of which initially consists of one base relation. The algorithm then calculates ranks for each pair of join trees and combines the pair with lowest rank to a single join tree. In addition to the originally proposed rank function *MinCard*, which minimizes the sizes of intermediate results, we use the C_{mm} cost function for ranking pairs of join trees, thus making GOO aware of indexes.

We also implemented the "BSizePP" heuristics proposed by Bruno et al. [6], which is based on Greedy Operator Ordering but takes index-nested-loop joins into account explicitly. BSizePP starts with the same set of trees as GOO and also picks two join trees to combine based on an index-aware rank function (again we use C_{mm} for ranking). Additional plans are explored by applying the following two transformations each time a pair of join trees has been selected:

1. *Push* each tree inside the other to correct mistakes that the greedy nature of the approach might have caused in previous steps.
2. *Pull* leaves out of the resulting tree to allow for index joins that might become cheaper now due to a decreased input cardinality.

The cheapest of the generated alternatives is selected as the join tree for the set of relations it contains. This process repeats until all trees are combined to a single remaining solution covering the complete set of relations.

Quickpick-1000 as well as the two deterministic heuristics can produce bushy plans, but obviously only explore parts of the search space. All algorithms in this experiment internally use the PostgreSQL cardinality estimates to compute a query plan, for which we compute the "true" cost using the true cardinalities.

The results of optimizing the 113 JOB queries using the aforementioned algorithms are summarized in Table 4. Apart from the strong impact of cardinality misestimates we identify the following factors influencing the quality of plans generated by an algorithm:

- *exploration depth*: Fully examining the search space using DP yields better plans than using heuristics (especially if many indexes are available).
- *index-awareness*: Algorithms that take indexes into account during join ordering, such as BSizePP and GOO/MinCost, outperform GOO/MinCard, which ignores indexes and orders joins solely by reducing the size of intermediate results.
- *exploration methodology*: Additional indexes add a few very good plans to the search space, which are less likely to be discovered by a randomized approach like Quickpick compared to systematic exploration as performed by GOO and BSizePP.

The push and pull transformations introduced by BSizePP have little effect compared to the index-aware GOO/MinCost.

To summarize, our results indicate that enumerating all bushy trees exhaustively offers moderate but not insignificant performance benefits in comparison to algorithms that enumerate only a subset of the search space. The performance potential from good cardinality estimates is certainly much larger. However, given the existence of exhaustive enumeration algorithms that can find the optimal solution for queries with dozens of relations very quickly (e.g., [36, 13]),

Table 4 Comparison of exhaustive dynamic programming with the Quickpick-1000 [49] (best of 1000 random plans), the Greedy Operator Ordering [12], and BSizePP [6]. All costs are normalized by the optimal plan of that index configuration.

	PK indexes						PK + FK indexes					
	PostgreSQL estimates			true cardinalities			PostgreSQL estimates			true cardinalities		
	median	95%	max	median	95%	max	median	95%	max	median	95%	max
Dynamic Programming	1.03	1.69	4.79	1.00	1.00	1.00	1.42	15.3	35.3	1.00	1.00	1.00
Quickpick-1000	1.05	7.99	42.4	1.00	1.08	1.21	2.10	77.6	3303	1.08	9.82	26.2
GOO/MinCard	1.01	1.85	3.42	1.01	1.37	1.92	1.84	34.1	766	1.55	28.6	766
GOO/MinCost	1.13	1.84	2.36	1.06	1.43	1.97	2.04	20.3	136	1.21	4.69	21.0
BSizePP	1.16	2.40	3.41	1.04	1.39	1.93	2.03	21.1	126	1.15	4.66	21.0

using heuristics or disabling bushy trees should only be necessary for queries with a large number of joins.

7 Join Ordering By Example

Up to now, when looking at plan quality, we mostly showed aggregated statistics. In order to get a better understanding of the effect of cardinality misestimation on plan quality, this section looks at two particular queries in detail. We use PostgreSQL as the source of cardinality estimates and the two query variants 13d and 13a as examples. The queries compute the ratings and release dates of movies produced by US (respectively, German) production companies. The two variants have the same structure (cf. Fig. 3) and join graph (cf. Fig. 4). The only difference is the predicate on `cn.country_code`, i.e., the base table selection on the `company_name` table[10]. In the remainder of this section we call query 13a the *German query* and 13d the *US query*.

As Fig. 6 shows, both queries exhibit the typical trend of growing underestimates as the number of joins increases. The base table predicate estimates, including the two different `cn.country_code` selections, are very close to the true cardinalities. PostgreSQL correctly estimates that the `company_name` table contains more than 8 times as many US companies than German companies. As a result, all intermediate results containing the `company_name` table differ by a similar factor in the two query variants. Interestingly, however, the PostgreSQL estimates lead to the same plan, which is shown in Fig. 13, for both variants.

To find out how good this plan is, we re-optimized both queries using the true cardinalities and obtained two different plans (Fig. 14 and 15 respectively). Comparing those three plans, we find that the PostgreSQL plan is quite good for the US query, as its true cost is only 35% higher than the cost of the optimal plan. Querying for the German movies using the same plan, however, is 3 times as expensive as the optimal plan.

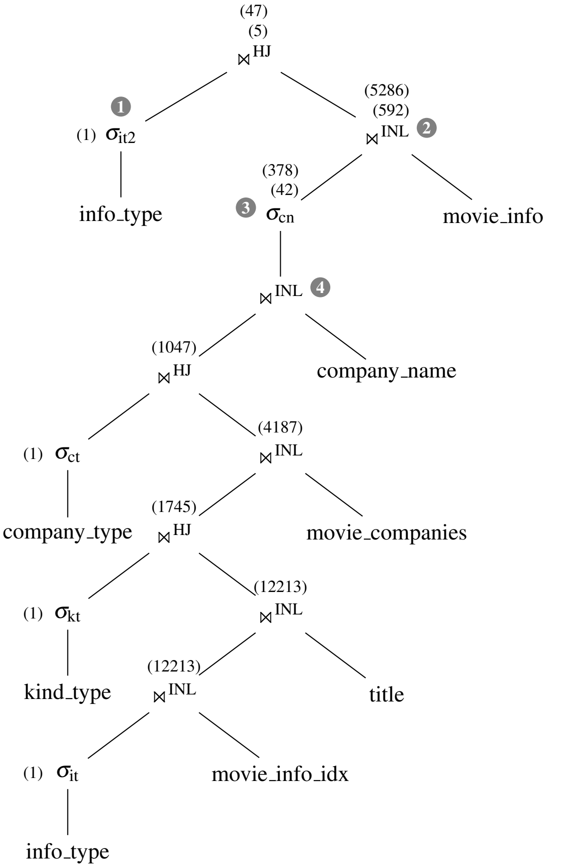


Fig. 13 Plan for both queries using PostgreSQL's estimates. This plan is fairly good for the US query, but not for the German query. The numbers in parentheses represent the estimated sizes of the respective intermediate results.

The plan quality difference of the two variants (the suboptimality of 1.35 vs. 3.05) is surprising given the fact that both queries have similarly large estimation errors and the only difference between the two queries is one predicate that is estimated accurately in both cases. To understand this result, one has to look at the different plans carefully. The optimal plan for the US query, which is shown in Fig. 14, is fairly similar to the one selected by PostgreSQL. The main difference is that index-nested-loop joins are more common due to underestimated intermediate results. However, none of these decisions have disastrous impact on the quality of

[10] Since the two query variants only differ in a constant within a selection predicate, they could be executed using the same prepared statement. Not statically knowing all constants statically presents additional, important, and well-researched challenges. However, we do not consider prepared statements in this work and always send the full query text.

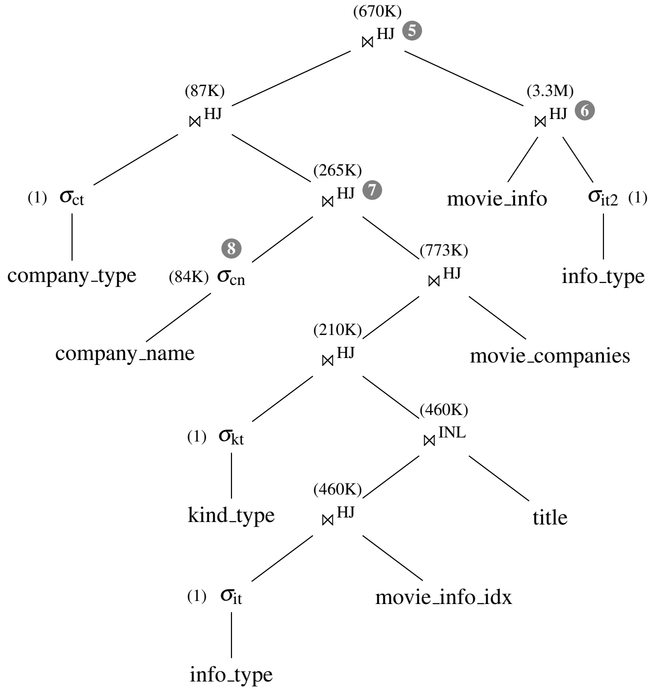


Fig. 14 Optimal plan for the US query using true cardinalities. The numbers in parentheses represent the true cardinalities of the respective intermediate results.

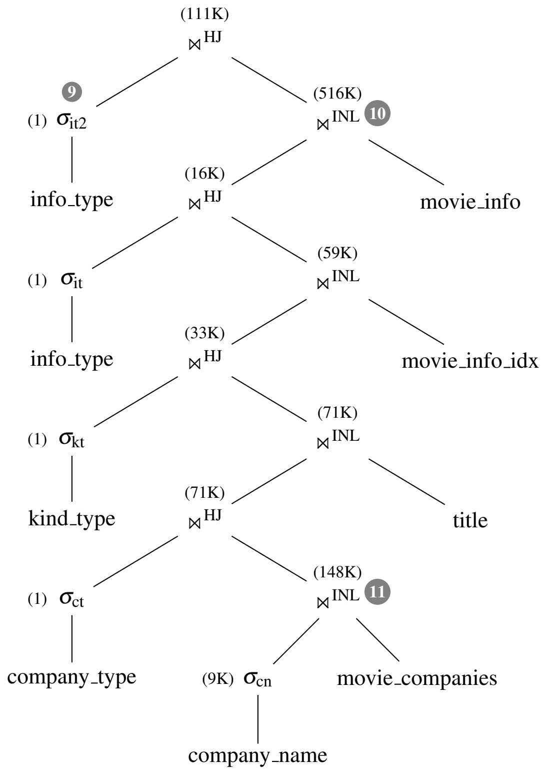


Fig. 15 Optimal plan for the German query using true cardinalities. The numbers in parentheses represent the true cardinalities of the respective intermediate results.

the plan. As Fig. 15 shows, the optimal plan for the German query, on the other hand, has a completely different structure than the one selected by PostgreSQL. In the following we identify three major differences between the plans and their impact on plan quality:

For the relatively few movies produced by German companies, using the index to join those movies with the `movie_info` table (❷ and ❿) and afterwards filtering for the release date information (❶ and ❾) is the right decision. However, performing this index-lookup for each of the many movies produced by US companies is no longer the cheapest solution. Instead, it would have been cheaper to first filter the `movie_info` for the release dates (❻) and afterwards join those with the US movies (❺).

Even more costly is the decision to index join all the movies rating information with the names of the involved companies (❹) and performing the required selection for German (respectively, US) companies (❸) on top of this join. Due to the severe underestimation of the left-hand intermediate result (1,047 instead of 303K for both queries), the index-nested-loop join looks much cheaper than it actually is (cf. ❼ and ❽).

Finally, even though the estimates for the base table selection on `company_name` are almost perfect for both queries (the maximum q-error is 1.03), PostgreSQL fails to detect the cheap join of the filtered German `company_name`s with the `movie_companies` (⓫) and hence does not use it as the bottom most join. Again the exponentially growing, systematic underestimation of intermediate cardinalities heavily distorts the optimizer's view: Many intermediate results for larger sets of relations are estimated to be much smaller than the number of German production companies, causing the optimizer to favor such subexpressions. This is the main cause for the plan in Fig. 13 being significantly worse to answer the query for German movies.

While these explanations are anecdotal since they only concern two queries, we have seen similar phenomena more frequently:

- In many cases misestimations cancel out each other—at least partially.
- Sometimes even large estimation errors have no influence on the optimality of a join ordering for a certain subexpression.
- In other cases, already slightly misestimated cardinalities may lead to large differences in plan quality.

8 Disk-based Experiments

While main memory databases are becoming widespread, large databases are still often stored on disk. Thus we now widen our investigation to include disk I/O cost as well. Our system has a hardware controller implementing RAID5,

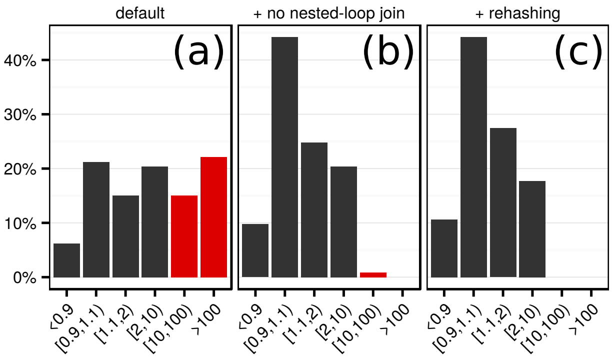


Fig. 16 Disk version of the experiments in Fig. 8. Slowdown distribution when using PostgreSQL estimates w.r.t. using true cardinalities – now on disk. On disk, the penalty for avoidable nested-loop joins (a, default PostgreSQL) is much higher. Even with this and hash join performance improved (cases b,c), more queries are 2-10 times slower than optimal, compared to the main memory case.

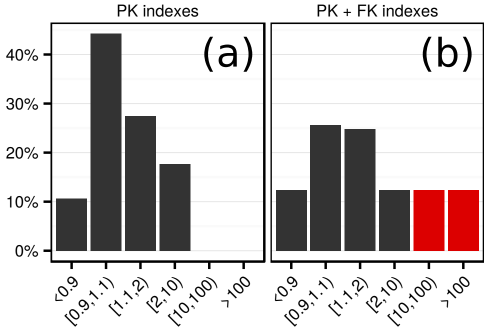


Fig. 17 Disk version of the experiments in Fig. 9. Slowdown distribution when using PostgreSQL estimates w.r.t. using true cardinalities. Using PK+FK indexes with wrong estimates on disk causes 10% of the queries to be slower than optimal by a factor of 100 or more; something that does not occur in main-memory.

which, using magnetic disks, achieves a read throughput of around 900 MB/s.

Due to being a real-world data set, we cannot simply increase the data size of the IMDB database (as would be possible with most synthetic benchmarks). In order to be able to run disk-based experiments, we instead decrease the PostgreSQL buffer pool to only 16 MB[11]. Additionally, we modified our JOB benchmark driver to flush the Linux (file) system buffer cache[12] before running each individual query. We increased the query timeout to 10 minutes. Repeating our experiments of Fig. 8, 9 and 10 under these conditions lasted roughly one week, and resulted in Fig. 16, 17, and 18.

8.1 Query Execution Engine

We now investigate the effects of the improvements to the query execution engine introduced in Section 5.2. Comparing Fig. 8a with Fig. 16a we see that the penalty for avoidable nested-loop joins is much higher on disk. Further, even when these are avoided and hash join performance is improved (cases a and c), significantly more queries are 2-10 times slower-than-optimal on-disk when compared to the main-memory case. On-disk processing thus makes the absolute cost distribution more extreme than in main-memory: slow/bad query plans are further away in time than fast/optimal query plans.

8.2 Adding Foreign Key Indexes

Although worse than in main-memory, query performance on disk is, despite the errors in cardinality estimation, still quite good with only primary-key indexes available. As can be seen from the on-disk Fig. 17, adding foreign-key indexes widens the cost distribution even further. Using PK+FK indexes with wrong estimates on disk causes 24% of the queries to be more than 10x slower than optimal with 12% being even more than a factor of 100 worse; something that occurs rarely in main-memory.

An in-depth analysis of the plans of the queries that were at least a factor of 10 slower than optimal revealed two main causes for them being slow: First, cardinalities, especially of small subexpressions (2-way and 3-way joins), are sometimes overestimated, resulting in a hash join to be used in places where performing a few index lookups would have been significantly faster. The vast majority of plans, however, becomes that slow due to excessive use of indexes. The optimal plans for the complete workload use an overall of 669 index-nested-loop joins, whereas using the PostgreSQL estimates results in 780 index-nested-loop joins being planned. In disk-based settings, planning index-nested-loop joins is risky due to expensive random IO required for the index lookups.

Not only are there more index-nested-loop joins planned, often the same joins are performed, but in a different order. Because of undetected join-crossing correlations, the optimizer misses opportunities to eliminate irrelevant tuples early. Thus, we often observe plans performing several orders of magnitude more index lookups than the optimal plan.

8.3 Tuning the Cost Model for Disk IO

One of the main findings so far was that physical cost modeling has become relatively unimportant in join order optimization for main-memory query processing, and research should focus on improving cardinality estimation. Does this

[11] We also ran experiments with a 128 MB buffer pool where we observed results that lie between the in-memory and the small buffer pool configuration.

[12] `echo 3 > /proc/sys/vm/drop_caches`

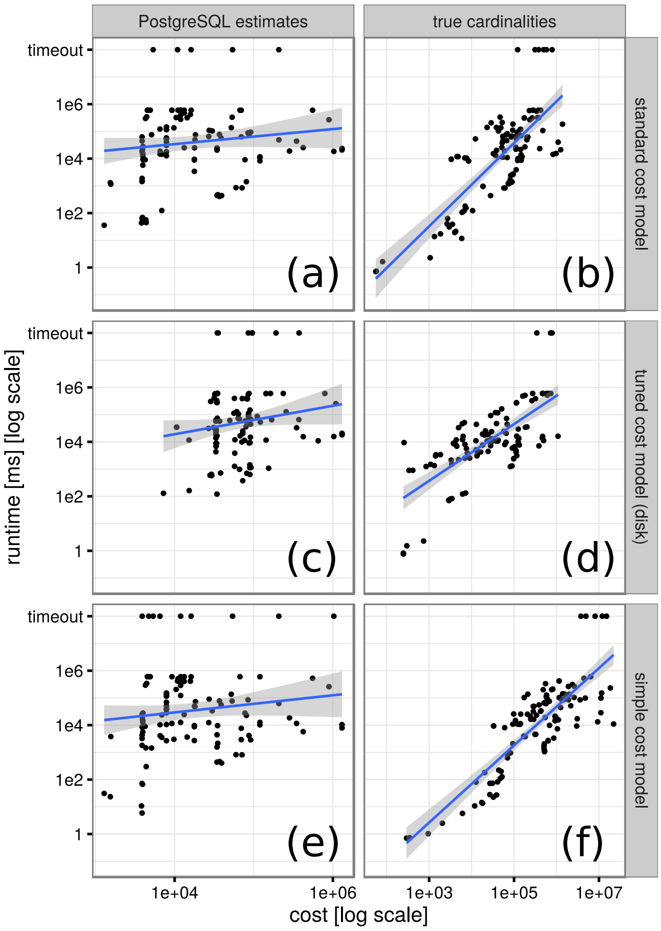


Fig. 18 Disk version of the experiments in Fig. 10. Note that the tuned cost model differs from Fig. 10, as we increase the relative cost of random to sequential I/O from the default factor of 4 to a more realistic factor of 100. On disk, true cardinalities are no longer perfect predictors of runtime (regardless of the tested cost model), and some high-cardinality queries time-out. Accurate estimates still correlate to runtime, but performance predictions are now very frequently off by an order of magnitude, giving rise to significant join-order sub-optimalities.

conclusion also hold for disk-based query processing? Fig. 18 which compares estimated cost with actual runtimes does confirm that cardinality estimation is indeed key to correctly predicting performance: even our trivial cost model that gets fed true cardinalities clearly correlates with on-disk query runtime, whereas with bad estimates (left side) there is no hope of good predictions. However, moving from main-memory (Fig. 10) to disk-based query processing does introduce significant errors to the predictions; on average one order of magnitude. In other words, even with perfect estimates, a join-order optimizer is likely to pick a suboptimal plan that could be an order of magnitude slower than optimal. Thus, for disk-based systems, a good cost model is more important than for systems purely operating in main-memory.

PostgreSQL's standard cost model assumes costs for fetching a random page to be 4x the cost of fetching pages sequentially. In reality, this factor is much higher, causing PostgreSQL to underestimate the cost of index lookups. We thus increased the cost of random IO to be 100x as expensive as sequential IO. The results of running the Join Order Benchmark with this tuned cost model are shown in subfigures c and d of Fig. 18. Using true cardinalities, tuning the cost model indeed improved the runtime prediction. About 63% of the queries are executed faster and 3 of the queries suffering timeout with the default cost model now execute successfully using the tuned cost model. However, the performance of 35% of the queries decreases significantly. Looking into the query plans, we find, that improvements are achieved due to less planning of index-nested-loop joins. The number of such joins dropped from 669 to 628 due to the tuned cost model. However, the cost model is still inaccurate in predicting query runtime, thus performance also worsens for a significant amount of queries. Finally, from the left hand subfigures in Fig. 18, we see that because of large errors in cardinality estimation, there is not much hope for good runtime predictions regardless of the cost model in use.

Summarizing the results of the on-disk experiments we can conclude that cost prediction gets harder in a disk-based environment and the influence of the cost model on plan quality is higher than in main memory. Besides the different characteristics between main memory and disk, further aspects, such as other storage devices, degree of parallelism and newer access methods [25] have an impact on the cost model. Thus, a cost model taking those factors into account may further increase the effect on plan quality. However, plan quality will generally still be dominated by the errors in cardinality estimation.

9 Related Work

Our cardinality estimation experiments show that systems which keep table samples for cardinality estimation predict single-table result sizes considerably better than those which apply the independence assumption and use single-column histograms [21]. We think systems should be adopting table samples as a simple and robust technique, rather than earlier suggestions to explicitly detect certain correlations [20] to subsequently create multi-column histograms [42] for these.

However, many of our JOB queries contain join-crossing correlations, which single-table samples do not capture, and where the current generation of systems still apply the independence assumption. There is a body of existing research work to better estimate result sizes of queries with join-crossing correlations, mainly based on join samples [18], possibly enhanced against skew (end-biased sampling [11], correlated samples [53], sampling-based query re-optimization [52], index-based join sampling [30]), using sketches [44] or graphical models [48]. This work confirms that without addressing join-crossing correlations, cardinality estimates deteriorate strongly with more joins [22], leading to both the over- and underestimation of result sizes (mostly the latter), so it would be positive if some of these techniques would be adopted by systems.

Another way of learning about join-crossing correlations is by exploiting query feedback, as in the LEO project [47], though there it was noted that deriving cardinality estimations based on a mix of exact knowledge and lack of knowledge needs a sound mathematical underpinning. For this, maximum entropy (MaxEnt [35,24]) was defined, though the costs for applying maximum entropy are high and have prevented its use in systems so far. We found that the performance impact of estimation mistakes heavily depends on the physical database design; in our experiments the largest impact is in situations with the richest designs. From the ROX [23] discussion in Section 4.4 one might conjecture that to truly unlock the potential of correctly predicting cardinalities for join-crossing correlations, we also need new physical designs and access paths.

Another finding in this paper is that the adverse effects of cardinality misestimations can be strongly reduced if systems would be "hedging their bets" and not only choose the plan with the cheapest expected cost, but take the probabilistic distribution of the estimate into account, to avoid plans that are marginally faster than others but bear a high risk of strong underestimation. There has been work both on doing this for cardinality estimates purely [37], as well as combining these with a cost model [2].

The problem with fixed hash table sizes for PostgreSQL illustrates that cost misestimation can often be mitigated by making the runtime behavior of the query engine more "performance robust". This can simply mean that operators do not use estimates in their implementation (e.g., [27,29]). More advanced techniques for making systems more adaptive include dynamically switch sides in a join or between hashing and sorting (GJoin [16]), switch between sequential scan and index lookup (smooth scan [5]), adaptively reordering join pipelines during query execution [31], or change aggregation strategies at runtime depending on the actual number of group-by values [38] or partition-by values [3].

A radical approach is to move query optimization to runtime, when actual value-distributions become available [40, 10]. However, runtime techniques typically restrict the plan search space to limit runtime plan exploration cost, and sometimes come with functional restrictions such as to only consider (sampling through) operators which have pre-created indexed access paths (e.g., ROX [23]).

Our experiments with the second query optimizer component besides cardinality estimation, namely the cost model, suggest that tuning cost models provides less benefits than improving cardinality estimates, and in a main-memory setting even an extremely simple cost-model can produce satisfactory results. This conclusion resonates with some of the findings in [51] which sets out to improve cost models but shows major improvements by refining cardinality estimates with additional sampling. In a disk-based setting, more accurate cost models have more impact and can improve query performance by an order of magnitude, but even this effect is generally overshadowed by the large cardinality estimation errors.

For testing the final query optimizer component, plan enumeration, we borrowed in our methodology from the Quickpick method used in randomized query optimization [49] to characterize and visualize the search space. Another well-known search space visualization method is Picasso [19], which visualizes query plans as areas in a space where query parameters are the dimensions. Interestingly, [49] claims in its characterization of the search space that good query plans are easily found, but our tests indicate that the richer the physical design and access path choices, the rarer good query plans become.

Query optimization is a core database research topic with a huge body of related work, that cannot be fully represented in this section. After decades of work still having this problem far from resolved [33], some have even questioned it and argued for the need of optimizer application hints [7]. This paper introduces the Join Order Benchmark based on the highly correlated IMDB real-world data set and a methodology for measuring the accuracy of cardinality estimation. Its integration in systems proposed for testing and evaluating the quality of query optimizers [50,17,15,34] is hoped to spur further innovation in this important topic.

10 Conclusions

Throughout this paper, in which we look at Query Optimization Through the Looking Glass, we made observations, some of which were already part of published literature or database systems lore, and others new. In the following we list all of these numbered by the section in which they are made, with the most important conclusions in bold:

Section 3.1: **Estimate cardinalities by execution on samples.** Cardinality estimation by evaluating predicates on small samples (e.g., 1000 tuples) and extrapolating from these, is to be preferred over other options (e.g., keeping histograms), since execution on samples automatically captures any predicate correlations between table columns, and is capable of estimating any filter predicate. With large data volumes in analytical queries and fast CPUs available now, both the absolute and relative overhead of execution on samples during query optimization has dropped (in the past, this overhead made this technique less attractive). In a main-memory setting, sampling and existing index structures can even be used to detect join-crossing correlations [30]. One caveat is that execution on samples has a vulnerability for very low-cardinality predicates (of which the sample holds 0 instances).

Sections 3.2, 5.1, 5.4: **Focus research on cardinality estimation rather than cost models.** Cardinality estimation of joins is the most important problem in query optimization. The estimates of all tested commercial systems routinely yield large estimation errors. Improving the accuracy of cardinality estimates is much more important for query optimizer quality than improving the accuracy of cost models. We tested an ultra-simple cost model that just sums the estimated amounts of intermediate tuples produced in a query, and this simple model performs just as well as the complex cost model of PostgreSQL in main memory, even after tuning it. In contrast, errors in cardinality estimation have a heavy effect on optimization quality.

Section 3.2: **Underestimation is more common than overestimation.** The more joins a real-life query has, the more current optimizers will underestimate the cardinalities due to applying the independence assumption. This implies that real-life queries tend to look for the Honda Accord (correlated predicates on brand and model) rather for the Honda Mustang (anti-correlated) because queries tend to be posed with certain embedded domain knowledge about actual, existing, entities. Our observations on System A suggests a heuristic that replaces simply applying selectivity multiplication (mandated by the independence assumption) by a "dampening" method that nudges selectivity downwards more gracefully. This suggestion is still a heuristic; of course, estimation methods that effectively capture join-crossing correlations would be better, but will be much harder to devise, given the huge space of potential correlations to cover.

Section 3.3: **Traditional benchmarks are not good tests for join order optimization.** TPC-H, TPC-DS, and SSB all work on data sets where column values have uniform or (in case of TPC-DS) stepwise-uniform frequency distributions, and which almost completely lack both intra- and inter-table correlations. This trivializes cardinality estimation. Furthermore, the queries in these benchmarks have a low number of joins, and this paper has shown that the hardness of accurate cardinality estimation increases directly with the number of joins.

Sections 3.4, 7: Query Optimization is quirky business. Two wrongs often make a right in query optimization. In multiple cases we illustrated the effect that multiple errors cancel each other out. This phenomenon makes analyzing and fixing query optimizer problems very frustrating because fixing one query will break another. Also, sometimes large estimation errors still lead the optimizer to find the right join order, whereas in other cases already slight misestimations have disastrous consequences.

Sections 4.1, 4.2, 8: One should use cost estimations in robust fashion. Rather than blindly picking the query plan with the lowest estimated cost, query optimizers should take the cardinality estimate error margins (or estimate probability density distribution) into account and avoid picking plans where the expected estimated cost is only slightly better but which run significant risk of being much slower than a robust alternative. In general, hash joins should be favored over nested-loop equi joins, because they are never much slower yet fall in a better complexity class. A related principle is never to rely on estimation for making decisions that can also be made at runtime, when the actual cardinalities are known (such as determining the amount of buckets in a hash-table, created for a hash join). As a final example, a rule that prefers to use index-nested-loops joins over hash joins is a robust choice in query optimization for main memory systems, since at worst there is only a small performance penalty in case hash joins would be better, whereas the upside can be large. In disk-based systems, where index-nested-loops joins lead to (slow) random I/O, this is not the case.

Sections 4.3, 6.1, 8: The richer the physical database infrastructure, the harder query optimization becomes. We observe the effect when adding unclustered foreign key indexes to the schema, which typically did lead to faster query times. However, with a richer schema, (i) the cost distribution of the plan space gets more diverse, often introducing a few (therefore hard-to-find) plans that are much faster and (ii) the slowdown experienced due to misestimations (compared to the optimal plan) is much higher than in the case without indexes or with only primary key indexes. The effect of (ii) is much larger in disk-based systems than in main memory systems.

Section 4.4: **Access paths for join-crossing correlations should be a research topic.** Correlations are a research frontier not only for correct estimation, but also in terms of devising new data structures, access paths, and execution algorithms. We discussed a DBLP co-authorship example query, where correctly predicting join-crossing anti-correlations is knowledge that cannot be leveraged, unless special physical database designs are deployed (in the example, table partitioning on a join-crossing attribute is needed). Thus, research in cost-estimation needs to be accompanied by research into new types of access paths.

Sections 6.2, 6.3: Superiority of exhaustive search. In rich schemas with FK indexes, exhaustive plan enumeration provides tangible benefits over more restricted strategies. Among these, restricting to zig-zag trees is better than considering only left-deep plans, which in turn is better than only considering right-deep plans. Heuristic strategies such as QuickPick, GOO, and BSizePP similarly find worse query plans than exhaustive search, especially in schemas with FK indexes, where index-aware approaches such as BSizePP perform better than the other heuristic approaches.

Acknowledgements We would like to thank Guy Lohman and the anonymous reviewers for their valuable feedback. We also thank Moritz Wilfer for his input in the early stages of this project.

References

1. Ahmed, R., Sen, R., Poess, M., Chakkappen, S.: Of snowstorms and bushy trees. PVLDB **7**(13), 1452–1461 (2014)
2. Babcock, B., Chaudhuri, S.: Towards a robust query optimizer: A principled and practical approach. In: SIGMOD, pp. 119–130 (2005)
3. Bellamkonda, S., Li, H.G., Jagtap, U., Zhu, Y., Liang, V., Cruanes, T.: Adaptive and big data scale parallel execution in Oracle. PVLDB **6**(11), 1102–1113 (2013)
4. Boncz, P.A., Neumann, T., Erling, O.: TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In: TPCTC, pp. 61–76 (2013)
5. Borovica-Gajic, R., Idreos, S., Ailamaki, A., Zukowski, M., Fraser, C.: Smooth scan: Statistics-oblivious access paths. In: ICDE, pp. 315–326 (2015)
6. Bruno, N., Galindo-Legaria, C.A., Joshi, M.: Polynomial heuristics for query optimization. In: ICDE, pp. 589–600 (2010)
7. Chaudhuri, S.: Query optimizers: time to rethink the contract? In: SIGMOD, pp. 961–968 (2009)
8. Chaudhuri, S., Narasayya, V.R., Ramamurthy, R.: Exact cardinality query optimization for optimizer testing. PVLDB **2**(1), 994–1005 (2009)
9. Colgan, M.: Oracle adaptive joins. `https://blogs.oracle.com/optimizer/entry/what_s_new_in_12c` (2013)
10. Dutt, A., Haritsa, J.R.: Plan bouquets: query processing without selectivity estimation. In: SIGMOD, pp. 1039–1050 (2014)
11. Estan, C., Naughton, J.F.: End-biased samples for join cardinality estimation. In: ICDE, p. 20 (2006)
12. Fegaras, L.: A new heuristic for optimizing large queries. In: DEXA, pp. 726–735 (1998)
13. Fender, P., Moerkotte, G.: Counter strike: Generic top-down join enumeration for hypergraphs. PVLDB **6**(14), 1822–1833 (2013)
14. Fender, P., Moerkotte, G., Neumann, T., Leis, V.: Effective and robust pruning for top-down join enumeration algorithms. In: ICDE, pp. 414–425 (2012)
15. Fraser, C., Giakoumakis, L., Hamine, V., Moore-Smith, K.F.: Testing cardinality estimation models in SQL Server. In: DBtest (2012)
16. Graefe, G.: A generalized join algorithm. In: BTW, pp. 267–286 (2011)
17. Gu, Z., Soliman, M.A., Waas, F.M.: Testing the accuracy of query optimizers. In: DBTest (2012)
18. Haas, P.J., Naughton, J.F., Seshadri, S., Swami, A.N.: Selectivity and cost estimation for joins based on random sampling. J Computer System Science **52**(3), 550–569 (1996)
19. Haritsa, J.R.: The Picasso database query optimizer visualizer. PVLDB **3**(2), 1517–1520 (2010)
20. Ilyas, I.F., Markl, V., Haas, P.J., Brown, P., Aboulnaga, A.: CORDS: automatic discovery of correlations and soft functional dependencies. In: SIGMOD, pp. 647–658 (2004)
21. Ioannidis, Y.E.: The history of histograms (abridged). In: VLDB, pp. 19–30 (2003)
22. Ioannidis, Y.E., Christodoulakis, S.: On the propagation of errors in the size of join results. In: SIGMOD (1991)
23. Kader, R.A., Boncz, P.A., Manegold, S., van Keulen, M.: ROX: run-time optimization of XQueries. In: SIGMOD, pp. 615–626 (2009)
24. Kaushik, R., Ré, C., Suciu, D.: General database statistics using entropy maximization. In: DBPL, pp. 84–99 (2009)
25. Kester, M.S., Athanassoulis, M., Idreos, S.: Access path selection in main-memory optimized data systems: Should I scan or should I probe? In: SIGMOD (2017)
26. Lang, H., Mühlbauer, T., Funke, F., Boncz, P.A., Neumann, T., Kemper, A.: Data Blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In: SIGMOD, pp. 311–326 (2016)
27. Leis, V., Boncz, P., Kemper, A., Neumann, T.: Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In: SIGMOD (2014)
28. Leis, V., Gubichev, A., Mirchev, A., Boncz, P.A., Kemper, A., Neumann, T.: How good are query optimizers, really? PVLDB **9**(3), 204–215 (2015)
29. Leis, V., Kundhikanjana, K., Kemper, A., Neumann, T.: Efficient processing of window functions in analytical SQL queries. PVLDB **8**(10) (2015)
30. Leis, V., Radke, B., Gubichev, A., Kemper, A., Neumann, T.: Cardinality estimation done right: Index-based join sampling. In: CIDR (2017)
31. Li, Q., Shao, M., Markl, V., Beyer, K.S., Colby, L.S., Lohman, G.M.: Adaptively reordering joins during query execution. In: ICDE, pp. 26–35 (2007)
32. Liu, F., Blanas, S.: Forecasting the cost of processing multi-join queries via hashing for main-memory databases. In: SoCC, pp. 153–166 (2015)
33. Lohman, G.: Is query optimization a solved problem? `http://wp.sigmod.org/?p=1075` (2014)
34. Mackert, L.F., Lohman, G.M.: R* optimizer validation and performance evaluation for local queries. In: SIGMOD, pp. 84–95 (1986)
35. Markl, V., Megiddo, N., Kutsch, M., Tran, T.M., Haas, P.J., Srivastava, U.: Consistently estimating the selectivity of conjuncts of predicates. In: VLDB, pp. 373–384 (2005)
36. Moerkotte, G., Neumann, T.: Dynamic programming strikes back. In: SIGMOD, pp. 539–552 (2008)
37. Moerkotte, G., Neumann, T., Steidl, G.: Preventing bad plans by bounding the impact of cardinality estimation errors. PVLDB **2**(1), 982–993 (2009)
38. Müller, I., Sanders, P., Lacurie, A., Lehner, W., Färber, F.: Cache-efficient aggregation: Hashing is sorting. In: SIGMOD, pp. 1123–1136 (2015)
39. Neumann, T.: Query simplification: graceful degradation for join-order optimization. In: SIGMOD, pp. 403–414 (2009)
40. Neumann, T., Galindo-Legaria, C.A.: Taking the edge off cardinality estimation errors using incremental execution. In: BTW, pp. 73–92 (2013)
41. O'Neil, P.E., O'Neil, E.J., Chen, X., Revilak, S.: The star schema benchmark and augmented fact table indexing. In: TPCTC, pp. 237–252 (2009)
42. Poosala, V., Ioannidis, Y.E.: Selectivity estimation without the attribute value independence assumption. In: VLDB, pp. 486–495 (1997)
43. Pöss, M., Nambiar, R.O., Walrath, D.: Why you should run TPC-DS: A workload analysis. In: PVLDB, pp. 1138–1149 (2007)
44. Rusu, F., Dobra, A.: Sketches for size of join estimation. TODS **33**(3) (2008)
45. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: SIGMOD, pp. 23–34 (1979)
46. Steinbrunn, M., Moerkotte, G., Kemper, A.: Heuristic and randomized optimization for the join ordering problem. VLDB J. **6**(3), 191–208 (1997)
47. Stillger, M., Lohman, G.M., Markl, V., Kandil, M.: LEO - DB2's learning optimizer. In: VLDB, pp. 19–28 (2001)
48. Tzoumas, K., Deshpande, A., Jensen, C.S.: Lightweight graphical models for selectivity estimation without independence assumptions. PVLDB **4**(11), 852–863 (2011)

49. Waas, F., Pellenkoft, A.: Join order selection - good enough is easy. In: BNCOD, pp. 51–67 (2000)

50. Waas, F.M., Giakoumakis, L., Zhang, S.: Plan space analysis: an early warning system to detect plan regressions in cost-based optimizers. In: DBTest (2011)

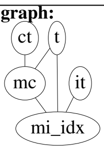
51. Wu, W., Chi, Y., Zhu, S., Tatemura, J., Hacigümüs, H., Naughton, J.F.: Predicting query execution time: Are optimizer cost models really unusable? In: ICDE, pp. 1081–1092 (2013)

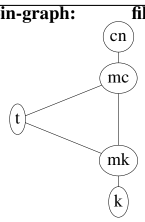
52. Wu, W., Naughton, J.F., Singh, H.: Sampling-based query re-optimization. In: SIGMOD (2016)

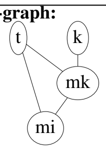
53. Yu, F., Hou, W., Luo, C., Che, D., Zhu, M.: CS2: a new database synopsis for query estimation. In: SIGMOD, pp. 469–480 (2013)

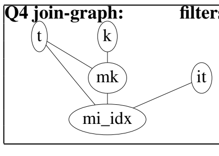
A Appendix: Detailed Query Descriptions

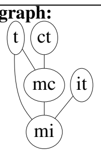
JOB consists of 113 multi-join *query variants* based on 33 *query structures*. The query variants derive from the same query structure differ only in their filter predicates; which consist of a series of *conjunctions*. The join-relationships connect the tables through *multiple* join relations, hence we draw the *join-graph* for each template on the left side. We use the alias from Table 1 as the tuple variable names for the joined relations. On the right side, we first list all filter predicates that are common to all variants of a query structure, followed by a box containing all additional filter predicates, one box for each query variant. On the bottom right, we list the projection columns (retrieved as MIN() aggregates). The query set is available online: `http://www-db.in.tum.de/~leis/qo/job.tgz`

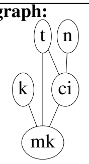
Q1 join-graph:  **filters:**	ct.kind = ’production companies’ mc.note NOT LIKE ’%(as Metro-Goldwyn-Mayer Pictures)%’
	1a: it.info = ’top 250 rank’ **1a:** (mc.note LIKE ’%(co-production)%’ OR mc.note LIKE ’%(presents)%’)
	1b: it.info = ’bottom 10 rank’ **1b:** t.production_year BETWEEN 2005 AND 2010
	1c: it.info = ’top 250 rank’ **1c:** (mc.note LIKE ’%(co-production)%’) **1c:** t.production_year > 2010
	1d: it.info = ’bottom 10 rank’ **1d:** t.production_year > 2000
	projections: mc.note t.title t.production_year

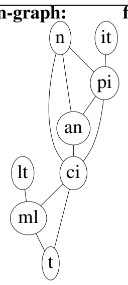
Q2 join-graph:  **filters:**	k.keyword = ’character-name-in-title’
	2a: cn.country_code =’[de]’
	2b: cn.country_code =’[nl]’
	2c: cn.country_code =’[sm]’
	2d: cn.country_code =’[us]’
	projections: t.title

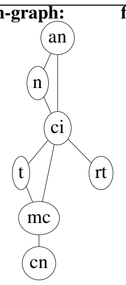
Q3 join-graph:  **filters:**	k.keyword LIKE ’%sequel%’
	3a: mi.info IN (’Sweden’, ’Norway’, ’Germany’, ’Denmark’, ’Swedish’, ’Denish’, ’Norwegian’, ’German’) **3a:** t.production_year > 2005
	3b: mi.info IN (’Bulgaria’) **3b:** t.production_year > 2010
	3c: mi.info IN (’Sweden’, ’Norway’, ’Germany’, ’Denmark’, ’Swedish’, ’Denish’, ’Norwegian’, ’German’, ’USA’, ’American’) **3c:** t.production_year > 1990
	projections: t.title

Q4 join-graph:  **filters:**	it.info = ’rating’ k.keyword LIKE ’%sequel%’
	4a: t.production_year > 2005
	4b: t.production_year > 2010
	4c: t.production_year > 1990
	projections: mi_idx.info t.title

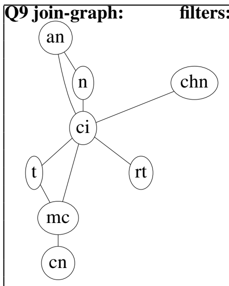
Q5 join-graph:  **filters:**	ct.kind = ’production companies’
	5a: mc.note LIKE ’%(France)%’ **5a:** mc.note LIKE ’%(theatrical)%’ **5a:** mi.info IN (’Sweden’, ’Norway’, ’Germany’, ’Denmark’, ’Swedish’, ’Denish’, ’Norwegian’, ’German’) **5a:** t.production_year > 2005
	5b: mc.note LIKE ’%(1994)%’ **5b:** mc.note LIKE ’%(USA)%’ **5b:** mc.note LIKE ’%(VHS)%’ **5b:** mi.info IN (’USA’, ’America’) **5b:** t.production_year > 2010
	5c: mc.note LIKE ’%(USA)%’ **5c:** mc.note NOT LIKE ’%(TV)%’ **5c:** mi.info IN (’Sweden’, ’Norway’, ’Germany’, ’Denmark’, ’Swedish’, ’Denish’, ’Norwegian’, ’German’, ’USA’, ’American’) **5c:** t.production_year > 1990
	projections: t.title

Q6 join-graph:  **filters:**	*(no common filter predicates)*
	6a: k.keyword = ’marvel-cinematic-universe’ **6a:** n.name LIKE ’%Downey%Robert%’ **6a:** t.production_year > 2010
	6b: k.keyword in (’superhero’, ’sequel’, ’second-part’, ’marvel-comics’, ’based-on-comic’, ’tv-special’, ’fight’, ’violence’) **6b:** n.name LIKE ’%Downey%Robert%’ **6b:** t.production_year > 2014
	6c: k.keyword = ’marvel-cinematic-universe’ **6c:** n.name LIKE ’%Downey%Robert%’ **6c:** t.production_year > 2014
	6d: k.keyword in (’superhero’, ’sequel’, ’second-part’, ’marvel-comics’, ’based-on-comic’, ’tv-special’, ’fight’, ’violence’) **6d:** n.name LIKE ’%Downey%Robert%’ **6d:** t.production_year > 2000
	6e: k.keyword = ’marvel-cinematic-universe’ **6e:** n.name LIKE ’%Downey%Robert%’ **6e:** t.production_year > 2000
	6f: k.keyword in (’superhero’, ’sequel’, ’second-part’, ’marvel-comics’, ’based-on-comic’, ’tv-special’, ’fight’, ’violence’) **6f:** t.production_year > 2000
	projections: k.keyword n.name t.title

Q7 join-graph:  **filters:**	it.info = ’mini biography’
	7a: an.name LIKE ’%a%’ **7a:** lt.link =’features’ **7a:** (n.gender=’m’ OR (n.gender = ’f’ **7a:** n.name LIKE ’B%’)) **7a:** n.name_pcode_cf BETWEEN ’A’ AND ’F’ **7a:** pi.note =’Volker Boehm’ **7a:** t.production_year BETWEEN 1980 AND 1995
	7b: an.name LIKE ’%a%’ **7b:** lt.link =’features’ **7b:** n.gender=’m’ **7b:** n.name_pcode_cf LIKE ’D%’ **7b:** pi.note =’Volker Boehm’ **7b:** t.production_year BETWEEN 1980 AND 1984
	7c: an.name is NOT NULL **7c:** (an.name LIKE ’%a%’ OR an.name LIKE ’A%’) **7c:** lt.link in (’references’, ’referenced in’, ’features’, ’featured in’) **7c:** (n.gender=’m’ OR (n.gender = ’f’ **7c:** n.name LIKE ’A%’)) **7c:** n.name_pcode_cf BETWEEN ’A’ AND ’F’ **7c:** pi.note is NOT NULL **7c:** t.production_year BETWEEN 1980 AND 2010
	projections: n.name t.title

Q8 join-graph:  **filters:**	*(no common filter predicates)*
	8a: ci.note =’(voice: English version)’ **8a:** cn.country_code =’[jp]’ **8a:** mc.note LIKE ’%(Japan)%’ **8a:** mc.note NOT LIKE ’%(USA)%’ **8a:** n.name LIKE ’%Yo%’ **8a:** n.name NOT LIKE ’%Yu%’ **8a:** rt.role =’actress’
	8b: ci.note =’(voice: English version)’ **8b:** cn.country_code =’[jp]’ **8b:** (mc.note LIKE ’%(2006)%’ OR mc.note LIKE ’%(2007)%’) **8b:** mc.note LIKE ’%(Japan)%’ **8b:** mc.note NOT LIKE ’%(USA)%’ **8b:** n.name LIKE ’%Yo%’ **8b:** n.name NOT LIKE ’%Yu%’ **8b:** rt.role =’actress’ **8b:** t.production_year BETWEEN 2006 AND 2007 **8b:** (t.title LIKE ’One Piece%’ OR t.title LIKE ’Dragon Ball Z%’)
	8c: cn.country_code =’[us]’ **8c:** rt.role =’writer’
	8d: cn.country_code =’[us]’ **8d:** rt.role =’costume designer’
	projections: an.name t.title

Q9 join-graph: **filters:**



cn.country_code = ’[us]’
n.gender = ’f’
rt.role = ’actress’

9a: ci.note in (’(voice)’, ’(voice: Japanese version)’, ’(voice) (uncredited)’, ’(voice: English version)’)
9a: mc.note is NOT NULL
9a: n.name LIKE ’%Ang%’
9a: t.production_year BETWEEN 2005 AND 2015

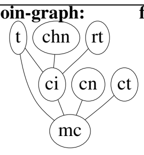
9b: ci.note = ’(voice)’
9b: mc.note LIKE ’%(200%)%’
9b: n.name LIKE ’%Angel%’
9b: t.production_year BETWEEN 2007 AND 2010

9c: ci.note in (’(voice)’, ’(voice: Japanese version)’, ’(voice) (uncredited)’, ’(voice: English version)’)
9c: n.name LIKE ’%An%’

9d: ci.note in (’(voice)’, ’(voice: Japanese version)’, ’(voice) (uncredited)’, ’(voice: English version)’)

projections: an.name chn.name t.title

Q10 join-graph: **filters:**



(no common filter predicates)

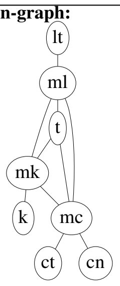
10a: ci.note LIKE ’%(uncredited)%’
10a: ci.note LIKE ’%(voice)%’
10a: cn.country_code = ’[ru]’
10a: rt.role = ’actor’
10a: t.production_year > 2005

10b: ci.note LIKE ’%(producer)%’
10b: cn.country_code = ’[ru]’
10b: rt.role = ’actor’
10b: t.production_year > 2010

10c: ci.note LIKE ’%(producer)%’
10c: cn.country_code = ’[us]’
10c: t.production_year > 1990

projections: chn.name t.title

Q11 join-graph: **filters:**



cn.country_code != ’[pl]’

11a: (cn.name LIKE ’%Film%’ OR cn.name LIKE ’%Warner%’)
11a: ct.kind =’production companies’
11a: k.keyword =’sequel’
11a: lt.link LIKE ’%follow%’
11a: mc.note IS NULL
11a: t.production_year BETWEEN 1950 AND 2000

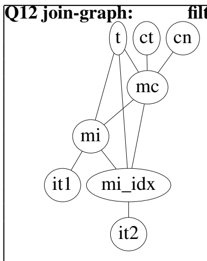
11b: (cn.name LIKE ’%Film%’ OR cn.name LIKE ’%Warner%’)
11b: ct.kind =’production companies’
11b: k.keyword =’sequel’
11b: lt.link LIKE ’%follows%’
11b: mc.note IS NULL
11b: t.production_year = 1998
11b: t.title LIKE ’%Money%’

11c: (cn.name LIKE ’20th Century Fox%’ OR cn.name LIKE ’Twentieth Century Fox%’)
11c: ct.kind is NOT NULL
11c: ct.kind != ’production companies’
11c: k.keyword in (’sequel’, ’revenge’, ’based-on-novel’)
11c: mc.note is NOT NULL
11c: t.production_year > 1950

11d: ct.kind is NOT NULL
11d: ct.kind != ’production companies’
11d: k.keyword in (’sequel’, ’revenge’, ’based-on-novel’)
11d: mc.note is NOT NULL
11d: t.production_year > 1950

projections: cn.name lt.link t.title

Q12 join-graph: **filters:**



cn.country_code = ’[us]’

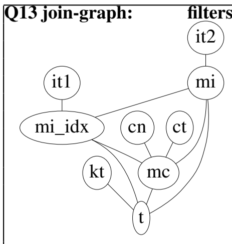
12a: ct.kind = ’production companies’
12a: it1.info = ’genres’
12a: it2.info = ’rating’
12a: mi.info in (’Drama’, ’Horror’)
12a: t.production_year BETWEEN 2005 AND 2008

12b: ct.kind is NOT NULL
12b: (ct.kind =’production companies’ OR ct.kind = ’distributors’)
12b: it1.info =’budget’
12b: it2.info =’bottom 10 rank’
12b: t.production_year > 2000
12b: (t.title LIKE ’Birdemic%’ OR t.title LIKE ’%Movie%’)

12c: ct.kind = ’production companies’
12c: it1.info = ’genres’
12c: it2.info = ’rating’
12c: mi.info in (’Drama’, ’Horror’, ’Western’, ’Family’)
12c: t.production_year BETWEEN 2000 AND 2010

projections: cn.name mi_idx.info t.title

Q13 join-graph: **filters:**



ct.kind = ’production companies’
it1.info = ’rating’
it2.info = ’release dates’
kt.kind = ’movie’

13a: cn.country_code =’[de]’

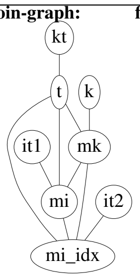
13b: cn.country_code =’[us]’
13b: t.title != ”
13b: (t.title LIKE ’%Champion%’ OR t.title LIKE ’%Loser%’)

13c: cn.country_code =’[us]’
13c: t.title != ”
13c: (t.title LIKE ’Champion%’ OR t.title LIKE ’Loser%’)

13d: cn.country_code =’[us]’

projections: mi.info mi_idx.info t.title

Q14 join-graph: **filters:**



it1.info = ’countries’
it2.info = ’rating’

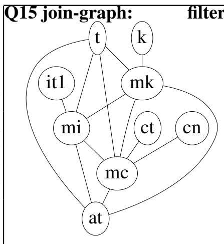
14a: k.keyword in (’murder’, ’murder-in-title’, ’blood’, ’violence’)
14a: kt.kind = ’movie’
14a: mi.info IN (’Sweden’, ’Norway’, ’Germany’, ’Denmark’, ’Swedish’, ’Denish’, ’Norwegian’, ’German’, ’USA’, ’American’)
14a: t.production_year > 2010

14b: k.keyword in (’murder’, ’murder-in-title’)
14b: kt.kind = ’movie’
14b: mi.info IN (’Sweden’, ’Norway’, ’Germany’, ’Denmark’, ’Swedish’, ’Denish’, ’Norwegian’, ’German’, ’USA’, ’American’)
14b: t.production_year > 2010
14b: (t.title LIKE ’%murder%’ OR t.title LIKE ’%Murder%’ OR t.title LIKE ’%Mord%’)

14c: k.keyword in (’murder’, ’murder-in-title’, ’blood’, ’violence’)
14c: k.keyword is NOT null
14c: kt.kind in (’movie’, ’episode’)
14c: mi.info IN (’Sweden’, ’Norway’, ’Germany’, ’Denmark’, ’Swedish’, ’Danish’, ’Norwegian’, ’German’, ’USA’, ’American’)
14c: t.production_year > 2005

projections: mi_idx.info t.title

Q15 join-graph: **filters:**



cn.country_code = ’[us]’
it1.info = ’release dates’
mi.note LIKE ’%internet%’

15a: mc.note LIKE ’%(200%)%’
15a: mi.info LIKE ’USA:% 200%’
15a: t.production_year > 2000

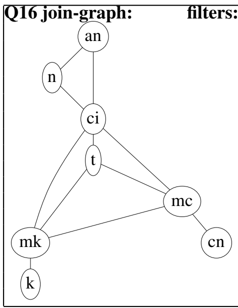
15b: cn.name = ’YouTube’
15b: mc.note LIKE ’%(200%)%’
15b: mi.info LIKE ’USA:% 200%’
15b: t.production_year BETWEEN 2005 AND 2010

15c: mi.info is NOT NULL
15c: (mi.info LIKE ’USA:% 199%’ OR mi.info LIKE ’USA:% 200%’)
15c: t.production_year > 1990

15d: t.production_year > 1990

projections: mi.info t.title

Q16 join-graph: **filters:**



cn.country_code = ’[us]’
k.keyword = ’character-name-in-title’

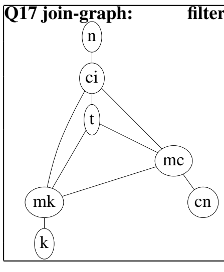
16a: t.episode_nr < 100
16a: t.episode_nr > = 50

16c: t.episode_nr < 100

16d: t.episode_nr < 100
16d: t.episode_nr > = 5

projections: an.name t.title

Q17 join-graph: **filters:**



k.keyword = ’character-name-in-title’

17a: cn.country_code =’[us]’
17a: n.name LIKE ’B%’

17b: n.name LIKE ’Z%’

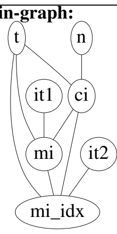
17c: n.name LIKE ’X%’

17d: n.name LIKE ’%Bert%’

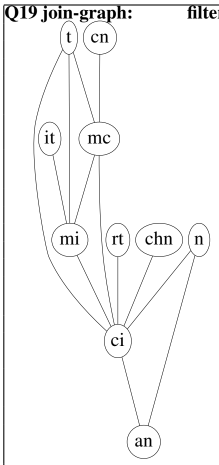
17e: cn.country_code =’[us]’

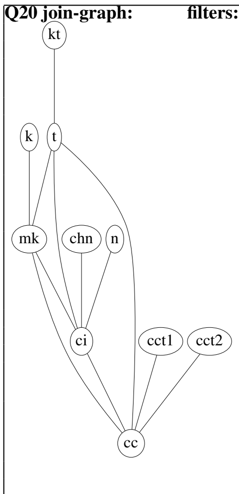
17f: n.name LIKE ’%B%’

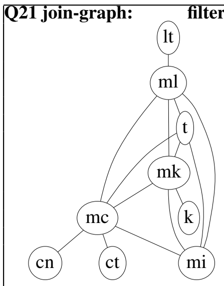
projections: n.name n.name

Q18 join-graph:	**filters:** *(no common filter predicates)*
	**18a:** ci.note in (’(producer)’, ’(executive producer)’) **18a:** it1.info = ’budget’ **18a:** it2.info = ’votes’ **18a:** n.gender = ’m’ **18a:** n.name LIKE ’%Tim%’
	18b: ci.note in (’(writer)’, ’(head writer)’, ’(written by)’, ’(story)’, ’(story editor)’) **18b:** it1.info = ’genres’ **18b:** it2.info = ’rating’ **18b:** mi.info in (’Horror’, ’Thriller’) **18b:** mi.note is NULL **18b:** n.gender = ’f’ **18b:** n.gender is NOT null **18b:** t.production_year BETWEEN 2008 AND 2014
	18c: ci.note in (’(writer)’, ’(head writer)’, ’(written by)’, ’(story)’, ’(story editor)’) **18c:** it1.info = ’genres’ **18c:** it2.info = ’votes’ **18c:** mi.info in (’Horror’, ’Action’, ’Sci-Fi’, ’Thriller’, ’Crime’, ’War’) **18c:** n.gender = ’m’
	projections: mi.info mi_idx.info t.title

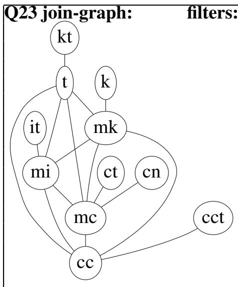
Q19 join-graph:	**filters:** cn.country_code = ’[us]’
	it.info = ’release dates’
	n.gender = ’f’
	rt.role = ’actress’

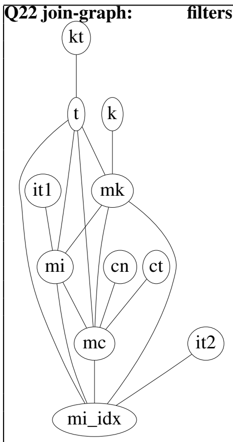
	**19a:** ci.note in (’(voice)’, ’(voice: Japanese version)’, ’(voice) (uncredited)’, ’(voice: English version)’) **19a:** mc.note is NOT NULL **19a:** mi.info is NOT null **19a:** (mi.info LIKE ’Japan:%200%’ OR mi.info LIKE ’USA:%200%’) **19a:** n.name LIKE ’%Ang%’ **19a:** t.production_year BETWEEN 2005 AND 2009
	19b: ci.note = ’(voice)’ **19b:** mc.note LIKE ’%(200%)%’ **19b:** mi.info is NOT null **19b:** (mi.info LIKE ’Japan:%2007%’ OR mi.info LIKE ’USA:%2008%’) **19b:** n.name LIKE ’%Angel%’ **19b:** t.production_year BETWEEN 2007 AND 2008 **19b:** t.title LIKE ’%Kung%Fu%PANDa%’
	19c: ci.note in (’(voice)’, ’(voice: Japanese version)’, ’(voice) (uncredited)’, ’(voice: English version)’) **19c:** mi.info is NOT null **19c:** (mi.info LIKE ’Japan:%200%’ OR mi.info LIKE ’USA:%200%’) **19c:** n.name LIKE ’%An%’ **19c:** t.production_year > 2000
	19d: ci.note in (’(voice)’, ’(voice: Japanese version)’, ’(voice) (uncredited)’, ’(voice: English version)’) **19d:** t.production_year > 2000
	projections: n.name t.title

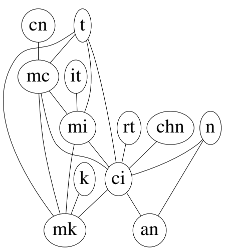
Q20 join-graph:	**filters:** cct1.kind = ’cast’ cct2.kind LIKE ’%complete%’ kt.kind = ’movie’
	**20a:** (chn.name LIKE ’%Tony%Stark%’ OR chn.name LIKE ’%Iron%Man%’) **20a:** chn.name NOT LIKE ’%Sherlock%’ **20a:** k.keyword in (’superhero’, ’sequel’, ’second-part’, ’marvel-comics’, ’based-on-comic’, ’tv-special’, ’fight’, ’violence’) **20a:** t.production_year > 1950
	20b: (chn.name LIKE ’%Tony%Stark%’ OR chn.name LIKE ’%Iron%Man%’) **20b:** chn.name NOT LIKE ’%Sherlock%’ **20b:** k.keyword in (’superhero’, ’sequel’, ’second-part’, ’marvel-comics’, ’based-on-comic’, ’tv-special’, ’fight’, ’violence’) **20b:** n.name LIKE ’%Downey%Robert%’ **20b:** t.production_year > 2000
	20c: chn.name is NOT NULL **20c:** (chn.name LIKE ’%man%’ OR chn.name LIKE ’%Man%’) **20c:** k.keyword in (’superhero’, ’marvel-comics’, ’based-on-comic’, ’tv-special’, ’fight’, ’violence’, ’magnet’, ’web’, ’claw’, ’laser’) **20c:** t.production_year > 2000
	projections: t.title

Q21 join-graph:	**filters:** cn.country_code != ’[pl]’ (cn.name LIKE ’%Film%’ OR cn.name LIKE ’%Warner%’) ct.kind = ’production companies’ k.keyword = ’sequel’ lt.link LIKE ’%follow%’ mc.note IS NULL
	**21a:** mi.info IN (’Sweden’, ’Norway’, ’Germany’, ’Denmark’, ’Swedish’, ’Denish’, ’Norwegian’, ’German’) **21a:** t.production_year BETWEEN 1950 AND 2000
	21b: mi.info IN (’Germany’, ’German’) **21b:** t.production_year BETWEEN 2000 AND 2010
	21c: mi.info IN (’Sweden’, ’Norway’, ’Germany’, ’Denmark’, ’Swedish’, ’Denish’, ’Norwegian’, ’German’, ’English’) **21c:** t.production_year BETWEEN 1950 AND 2010
	projections: cn.name lt.link t.title

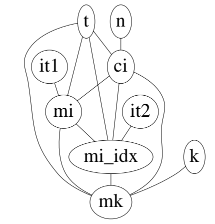
Q23 join-graph:	**filters:** cct.kind = ’complete+verified’
	cn.country_code = ’[us]’
	it.info = ’release dates’
	mi.note LIKE ’%internet%’

 kt t k it mk mi ct cn mc cct cc	**23a:** kt.kind in (’movie’) **23a:** mi.info is NOT NULL **23a:** (mi.info LIKE ’USA:% 199%’ OR mi.info LIKE ’USA:% 200%’) **23a:** t.production_year > 2000
	23b: k.keyword in (’nerd’, ’loner’, ’alienation’, ’dignity’) **23b:** kt.kind in (’movie’) **23b:** mi.info LIKE ’USA:% 200%’ **23b:** t.production_year > 2000
	23c: mi.info is NOT NULL **23c:** (mi.info LIKE ’USA:% 199%’ OR mi.info LIKE ’USA:% 200%’) **23c:** t.production_year > 1990
	projections: kt.kind t.title

Q22 join-graph:	**filters:** cn.country_code != ’[us]’ it1.info = ’countries’ it2.info = ’rating’ k.keyword in (’murder’, ’murder-in-title’, ’blood’, ’violence’) kt.kind in (’movie’, ’episode’)
 kt t k it1 mk mi cn ct mc it2 mi_idx	**22a:** mc.note LIKE ’%(200%)%’ **22a:** mc.note NOT LIKE ’%(USA)%’ **22a:** mi.info IN (’Germany’, ’German’, ’USA’, ’American’) **22a:** t.production_year > 2008
	22b: mc.note LIKE ’%(200%)%’ **22b:** mc.note NOT LIKE ’%(USA)%’ **22b:** mi.info IN (’Germany’, ’German’, ’USA’, ’American’) **22b:** t.production_year > 2009
	22c: mc.note LIKE ’%(200%)%’ **22c:** mc.note NOT LIKE ’%(USA)%’ **22c:** mi.info IN (’Sweden’, ’Norway’, ’Germany’, ’Denmark’, ’Swedish’, ’Danish’, ’Norwegian’, ’German’, ’USA’, ’American’) **22c:** t.production_year > 2005
	22d: mi.info IN (’Sweden’, ’Norway’, ’Germany’, ’Denmark’, ’Swedish’, ’Danish’, ’Norwegian’, ’German’, ’USA’, ’American’) **22d:** t.production_year > 2005
	projections: cn.name mi_idx.info t.title

Q24 join-graph:	**filters:** ci.note in (’(voice)’, ’(voice: Japanese version)’, ’(voice) (uncredited)’, ’(voice: English version)’) cn.country_code = ’[us]’ it.info = ’release dates’ mi.info is NOT null (mi.info LIKE ’Japan:%201%’ OR mi.info LIKE ’USA:%201%’) n.gender = ’f’ n.name LIKE ’%An%’ rt.role = ’actress’ t.production_year > 2010
 cn t mc it mi rt chn n k ci mk an	**24a:** k.keyword in (’hero’, ’martial-arts’, ’hAND-to-hAND-combat’)
	24b: cn.name = ’DreamWorks Animation’ **24b:** k.keyword in (’hero’, ’martial-arts’, ’hAND-to-hAND-combat’, ’computer-animated-movie’) **24b:** t.title LIKE ’Kung Fu PANDa%’
	projections: chn.name n.name t.title

Q25 join-graph: **filters:**



ci.note in ('(writer)', '(head writer)', '(written by)', '(story)', '(story editor)')
it1.info = 'genres'
it2.info = 'votes'

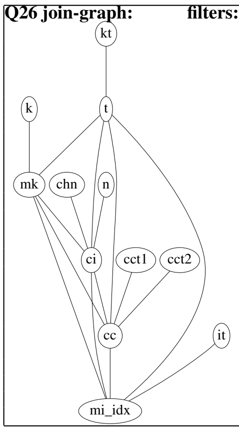
25a: k.keyword in ('murder', 'blood', 'gore', 'death', 'female-nudity')
25a: mi.info = 'Horror'
25a: n.gender = 'm'

25b: k.keyword in ('murder', 'blood', 'gore', 'death', 'female-nudity')
25b: mi.info = 'Horror'
25b: n.gender = 'm'
25b: t.production_year > 2010
25b: t.title LIKE 'Vampire%'

25c: k.keyword in ('murder', 'violence', 'blood', 'gore', 'death', 'female-nudity', 'hospital')
25c: mi.info in ('Horror', 'Action', 'Sci-Fi', 'Thriller', 'Crime', 'War')
25c: n.gender = 'm'

projections: mi.info mi_idx.info n.name t.title

Q26 join-graph: **filters:**



cct1.kind = 'cast'
cct2.kind LIKE '%complete%'
chn.name is NOT NULL
(chn.name LIKE '%man%' OR chn.name LIKE '%Man%')
it.info = 'rating'
kt.kind = 'movie'

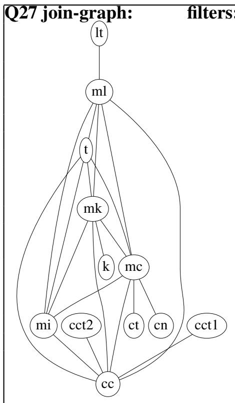
26a: k.keyword in ('superhero', 'marvel-comics', 'based-on-comic', 'tv-special', 'fight', 'violence', 'magnet', 'web', 'claw', 'laser')
26a: t.production_year > 2000

26b: k.keyword in ('superhero', 'marvel-comics', 'based-on-comic', 'fight')
26b: t.production_year > 2005

26c: k.keyword in ('superhero', 'marvel-comics', 'based-on-comic', 'tv-special', 'fight', 'violence', 'magnet', 'web', 'claw', 'laser')
26c: t.production_year > 2000

projections: chn.name mi_idx.info n.name t.title

Q27 join-graph: **filters:**



cn.country_code != '[pl]'
(cn.name LIKE '%Film%' OR cn.name LIKE '%Warner%')
ct.kind = 'production companies'
k.keyword = 'sequel'
lt.link LIKE '%follow%'
mc.note IS NULL

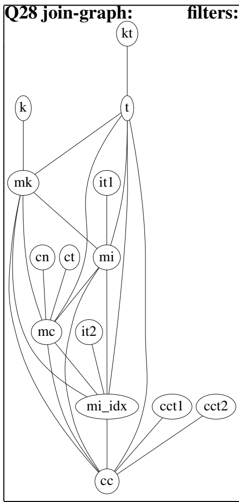
27a: cct1.kind in ('cast', 'crew')
27a: cct2.kind = 'complete'
27a: mi.info IN ('Sweden', 'Germany','Swedish', 'German')
27a: t.production_year BETWEEN 1950 AND 2000

27b: cct1.kind in ('cast', 'crew')
27b: cct2.kind = 'complete'
27b: mi.info IN ('Sweden', 'Germany','Swedish', 'German')
27b: t.production_year = 1998

27c: cct1.kind = 'cast'
27c: cct2.kind LIKE 'complete%'
27c: mi.info IN ('Sweden', 'Norway', 'Germany', 'Denmark', 'Swedish', 'Denish', 'Norwegian', 'German', 'English')
27c: t.production_year BETWEEN 1950 AND 2010

projections: cn.name lt.link t.title

Q28 join-graph: **filters:**



cn.country_code != '[us]'
it1.info = 'countries'
it2.info = 'rating'
k.keyword in ('murder', 'murder-in-title', 'blood', 'violence')
kt.kind in ('movie', 'episode')
mc.note LIKE '%(200%)%'
mc.note NOT LIKE '%(USA)%'

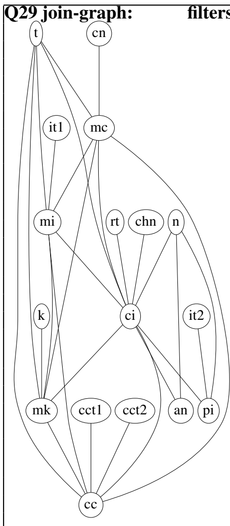
28a: cct1.kind = 'crew'
28a: cct2.kind != 'complete+verified'
28a: mi.info IN ('Sweden', 'Norway', 'Germany', 'Denmark', 'Swedish', 'Danish', 'Norwegian', 'German', 'USA', 'American')
28a: t.production_year > 2000

28b: cct1.kind = 'crew'
28b: cct2.kind != 'complete+verified'
28b: mi.info IN ('Sweden', 'Germany', 'Swedish', 'German')
28b: t.production_year > 2005

28c: cct1.kind = 'cast'
28c: cct2.kind = 'complete'
28c: mi.info IN ('Sweden', 'Norway', 'Germany', 'Denmark', 'Swedish', 'Danish', 'Norwegian', 'German', 'USA', 'American')
28c: t.production_year > 2005

projections: cn.name mi_idx.info t.title

Q29 join-graph: **filters:**



cct1.kind = 'cast'
cct2.kind = 'complete+verified'
cn.country_code = '[us]'
it1.info = 'release dates'
k.keyword = 'computer-animation'
n.gender = 'f'
n.name LIKE '%An%'
rt.role = 'actress'

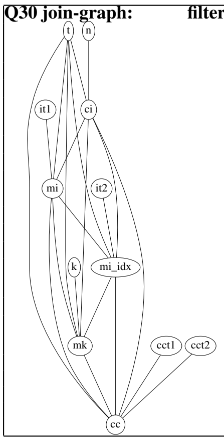
29a: chn.name = 'Queen'
29a: ci.note in ('(voice)', '(voice) (uncredited)', '(voice: English version)')
29a: it2.info = 'trivia'
29a: mi.info is NOT null
29a: (mi.info LIKE 'Japan:%200%' OR mi.info LIKE 'USA:%200%')
29a: t.production_year BETWEEN 2000 AND 2010
29a: t.title = 'Shrek 2'

29b: chn.name = 'Queen'
29b: ci.note in ('(voice)', '(voice) (uncredited)', '(voice: English version)')
29b: it2.info = 'height'
29b: mi.info LIKE 'USA:%200%'
29b: t.production_year BETWEEN 2000 AND 2005
29b: t.title = 'Shrek 2'

29c: ci.note in ('(voice)', '(voice: Japanese version)', '(voice) (uncredited)', '(voice: English version)')
29c: it2.info = 'trivia'
29c: mi.info is NOT null
29c: (mi.info LIKE 'Japan:%200%' OR mi.info LIKE 'USA:%200%')
29c: t.production_year BETWEEN 2000 AND 2010

projections: chn.name n.name t.title

Q30 join-graph: **filters:**



cct2.kind = 'complete+verified'
ci.note in ('(writer)', '(head writer)', '(written by)', '(story)', '(story editor)')
it1.info = 'genres'
it2.info = 'votes'
k.keyword in ('murder', 'violence', 'blood', 'gore', 'death', 'female-nudity', 'hospital')
n.gender = 'm'

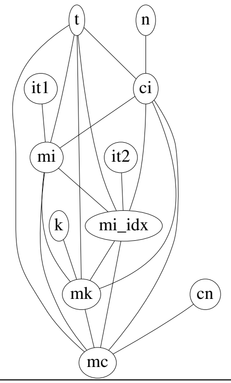
30a: cct1.kind in ('cast', 'crew')
30a: mi.info in ('Horror', 'Thriller')
30a: t.production_year > 2000

30b: cct1.kind in ('cast', 'crew')
30b: mi.info in ('Horror', 'Thriller')
30b: t.production_year > 2000
30b: (t.title LIKE '%Freddy%' OR t.title LIKE '%Jason%' OR t.title LIKE 'Saw%')

30c: cct1.kind = 'cast'
30c: mi.info in ('Horror', 'Action', 'Sci-Fi', 'Thriller', 'Crime', 'War')

projections: mi.info mi_idx.info n.name t.title

Q31 join-graph: **filters:**



ci.note in ('(writer)', '(head writer)', '(written by)', '(story)', '(story editor)')
cn.name LIKE 'Lionsgate%'
it1.info = 'genres'
it2.info = 'votes'
k.keyword in ('murder', 'violence', 'blood', 'gore', 'death', 'female-nudity', 'hospital')

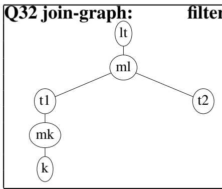
31a: mi.info in ('Horror', 'Thriller')
31a: n.gender = 'm'

31b: mc.note LIKE '%(Blu-ray)%'
31b: mi.info in ('Horror', 'Thriller')
31b: n.gender = 'm'
31b: t.production_year > 2000
31b: (t.title LIKE '%Freddy%' OR t.title LIKE '%Jason%' OR t.title LIKE 'Saw%')

31c: mi.info in ('Horror', 'Action', 'Sci-Fi', 'Thriller', 'Crime', 'War')

projections: mi.info mi_idx.info n.name t.title

Q32 join-graph: **filters:**



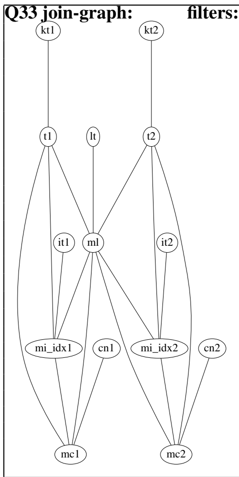
(no common filter predicates)

32a: k.keyword ='10,000-mile-club'

32b: k.keyword ='character-name-in-title'

projections: lt.link t1.title t2.title

Q33 join-graph: **filters:**	it1.info = ’rating’
	it2.info = ’rating’

 kt1 kt2 t1 lt t2 it1 ml it2 mi_idx1 cn1 mi_idx2 cn2 mc1 mc2	**33a:** cn1.country_code = ’[us]’ **33a:** kt1.kind in (’tv series’) **33a:** kt2.kind in (’tv series’) **33a:** lt.link in (’sequel’, ’follows’, ’followed by’) **33a:** t2.production_year BETWEEN 2005 AND 2008
	33b: cn1.country_code = ’[nl]’ **33b:** kt1.kind in (’tv series’) **33b:** kt2.kind in (’tv series’) **33b:** lt.link LIKE ’%follow%’ **33b:** t2.production_year = 2007
	33c: cn1.country_code != ’[us]’ **33c:** kt1.kind in (’tv series’, ’episode’) **33c:** kt2.kind in (’tv series’, ’episode’) **33c:** lt.link in (’sequel’, ’follows’, ’followed by’) **33c:** t2.production_year BETWEEN 2000 AND 2010
	projections: cn1.name cn2.name mi_idx1.info mi_idx2.info t1.title t2.title