10381 Summary and Abstracts Collection

Robust Query Processing

— Dagstuhl Seminar —

Goetz Graefe[1], Harumi Anne Kuno[2], Arnd Christian König[3], Volker Markl[4] and Kai-Uwe Sattler[5]

[1] Hewlett Packard Labs - Palo Alto, US
goetz.graefe@hp.com
[2] Hewlett Packard Labs - Palo Alto, US
harumi.kuno@hp.com
[3] Microsoft Research - Redmond, US
[4] TU Berlin, DE
volker.markl@tu-berlin.de
[5] TU Ilmenau, DE
kus@tu-ilmenau.de

Abstract. Dagstuhl seminar 10381 on robust query processing (held 19.09.10 - 24.09.10) brought together a diverse set of researchers and practitioners with a broad range of expertise for the purpose of fostering discussion and collaboration regarding causes, opportunities, and solutions for achieving robust query processing. The seminar strove to build a unified view across the loosely-coupled system components responsible for the various stages of database query processing. Participants were chosen for their experience with database query processing and, where possible, their prior work in academic research or in product development towards robustness in database query processing. In order to pave the way to motivate, measure, and protect future advances in robust query processing, seminar 10381 focused on developing tests for measuring the robustness of query processing. In these proceedings, we first review the seminar topics, goals, and results, then present abstracts or notes of some of the seminar break-out sessions. We also include, as an appendix, the robust query processing reading list that was collected and distributed to participants before the seminar began, as well as summaries of a few of those papers that were contributed by some participants.

Keywords. Robust query processing, adaptive query optimization, query execution, indexing, workload management, reliability, application availability

1 Motivation and Goals

In the context of data management, robustness is usually associated with recovery from failure, redundancy, disaster preparedness, etc. Robust query processing, on the other hand, is about robustness of performance and scalability. It is more than progress reporting or predictability. A system that predictably fails or obviously performs poorly is somewhat more useful than an unpredictable one, but it is not robust. This is comparable to an automobile that only starts in dry weather: it is predictable but not nearly as useful or robust as a car that starts in any weather.

Robust query processing performance has been a known problem for a long time. It also seems common to most or all database management systems and most or all installations. All experienced database administrators know of sudden disruptions of data center processing due to database queries performing poorly, including queries that had performed flawlessly or at least acceptably for days or weeks.

Some techniques are meant to alleviate problems of poor performance, e.g., automatic index tuning or statistics gathered and refreshed on-demand. However, they sometime exacerbate the problem. For example, insertion of a few new rows into a large table might trigger an automatic update of statistics, which uses a different sample than the prior one, which leads to slightly different histograms, which results in slightly different cardinality or cost estimates, which leads to an entirely different query execution plan, which might actually perform much worse than the prior one due to estimation errors. Such occasional "automatic disasters" are difficult to spot and usually require lengthy and expensive root cause analysis, often at an inconvenient time.

A frequent cause of unpredictable performance is that compile-time query optimization is liable to suffer from inaccuracy in cardinality estimation or in cost calculations. Such errors are common in queries with dozens of tables or views, typically generated by software for business intelligence or for mapping objects to relational databases. Estimation errors do not necessarily lead to poor query execution plans, but they do so often and at unpredictable times.

Other sources for surprising query performance are widely fluctuating workloads, conflicts in concurrency control, changes in physical database design, rigid resource management such as a fixed-size in-memory workspace for sorting, and, of course, automatic tuning of physical database design or of server parameters such as memory allocation for specific purposes such as sorting or index creation.

Numerous approaches and partial solutions have been proposed over the decades, from automatic index tuning, automatic database statistics, self-correcting cardinality estimation in query optimization, dynamic resource management, adaptive workload management, and many more. Many of them are indeed practical and promising, but there is no way of comparing the value of competing techniques (and they all compete at least for implementation engineers!) until a useful metric for query processing robustness has been defined. Thus, defining robustness as well as a metric for it is a crucial step towards making progress.

Such a metric can serve multiple purposes. The most mundane purpose might be regression testing, i.e., to ensure that progress, once achieved in a code base, is not lost in subsequent maintenance or improvement of seemingly unrelated code or functionality. The most public purpose might be to compare competing software packages in terms of their robustness in query processing performance and scalability as a complement to existing benchmarks that measure raw performance and scalability without regard to robustness.

2 Outcome and next steps

The seminar was well attended: 34 researchers (with 16 researchers from industry) from Europe, India, and North America actively explored metrics and tests in the context of physical database design, query optimization, query execution, and workload management. Participants investigated many approaches to measuring and testing robustness without being able to unify them into a single metric. It became clear, however, that continuous parameters such as sizes of tables and intermediate results are much more tractable than discrete parameters such as presence or absence of specific indexes.

At this time, we are pursuing multiple steps based on this seminar. First, several groups of participants are researching and authoring papers on robust query processing, its causes and appropriate metrics. Second, we have been invited to edit a special issue of a journal. Third, we have been invited to organize a panel on robust query processing in an international database conference. Fourth, we have applied for a follow-on seminar in Dagstuhl that will focus on continuous parameters (such as table size), on turning discrete parameters (such as existence of a specific index) into a continuous one, and on scalability and problems in high parallel query processing including cloud servers.

3 Working Agenda

Each day had a theme related to seminar goals, and several breakout sessions were organized where the seminar was split in smaller groups. The topics of these sessions were structured along the main areas of robustness in query processing: test suite design, query optimization, query execution, physical database design, and system context.

The goal of the first day's breakout sessions was test suite design, i.e. which aspects of robust query processing should be tested and which metrics are appropriate. The second day's session were devoted to robustness issues in query optimization. Measuring the contribution of query execution and access methods to robust query processing was the goal of day 3. On the fourth day, two topics were addressed in parallel. In the first track of sessions, robustness of physical database design tools such as index selection tools and design advisors was discussed. The second track of sessions addressed a broader system context by discussing the interaction of various system components in the relationship to robustness.

3.1 Day 1: Definitions and tests

Goals for the day:

– reach a common background and understanding,

– establish mode & mood for collaboration,
– draft regression tests for solved and unsolved problems in robust query processing,
– initialize possible publications.

Session 1.1: Stories from the real world

Session goals:
- initialize the seminar,
- share anecdotes and stories,
- define & scope the problem of robust query processing,
- separate solved and unsolved problems,
- define scope, outcome, and process for the seminar.

Scope: problems (not solutions), real-world importance.

Preparation: presentation slides with scope (both included & excluded topics) – preliminary analysis of prepared anecdotes.

Discussion: what went wrong in those cases, what was the root cause, what partial or complete solutions are known, can those solutions be automated – what is robust query processing? – what problems have “obvious” or known solutions, what problems remain?

Session 1.2: Regression tests

Session goal: share understanding of and appreciation for regression testing in the context of robust query processing and good regression testing.

Scope: testing methodology.

Preparation: presentation slides with an example regression test for a robust technique – anecdote of a failure due to lacking regression tests – example scoring based on visualization of index nested loops join.

Discussion: what characterizes a good regression test suite (efficient, effective, repeatable, portable...) – what is the role of regression testing in research and development of robust query processing – if the “obvious” solutions were implemented, how would they be protected by testing? – how to talk about and how to visualize robustness and the lack thereof.

Session 1.3: Break out: test suite design

Session goal: design test suites, including specific SQL tables and queries, for problems with obvious or known solutions and for unsolved problems – broken up into scans & data access, join order and join algorithms, nested iteration, updates & utilities; concurrency control & multi-programming level, etc.

Scope: tests (not solutions) for “solved” and “unsolved” problems in robust query processing.

Preparation: a sample test suite as SQL script – a sample paper with a specific to-do list starting with electing a scribe & paper coordinator. Survey of existing test suites.

Discussion: What aspects of "robust query processing" should be tested, which metrics are appropriate for these aspects, and how can one compare metrics? Can we extend any existing test suites to evaluate these aspects? What should we consider when designing the test suites? (E.g., index interactions, workload variability/volatility, accuracy of estimates, accuracy of statistics, how to generate queries, how to generate data.)

Session 1.4: Review of test suites

Session goal: ensure a common understanding of draft test suites – focus on tests & metrics versus techniques & solutions – refine drafts as appropriate.

Preparation: Write up characterization of each draft test suite and list systems/authors who may apply that test suite, as well as an evaluation of target application of the test suite.

Discussion: Practical concerns and ideas about potential learnings raised by each test suite. Evaluate each test suite in terms of how it compares to existing test suites, what one might learn from it, why it might be difficult to implement, how it might be difficult to interpret results, and how it might be simplified.

3.2 Day 2: Query optimization

Goal for the day: define regression tests to protect advances & delineate limitations of query optimization and its contributions to robust query processing – initialize possible publications.

Session 2.1: Traditional query optimization

Session goals:
- ensure a shared understanding of risks in query optimization & its components,
- identify role & risks of nested iteration,
- list problems to be solved by appropriate query optimization,
- separate metrics from techniques, "plan quality" from specific query optimization or query execution techniques,
- isolate & classify common assumptions.

Scope: focus on "simple" compile-time query optimization – exclude dynamic query execution plans, user-defined functions, hints for physical database design, etc.

Preparation: presentation slides with components of query processing, query optimization (cardinality estimation, cost calculation, plan repertoire, plan search & heuristics), query execution (plan distribution, pipelining & granularity, partitioning, individual algorithms, resource management), access methods (index format, concurrency control).

Discussion: need to test cardinality estimation, cost calculation, plan repertoire, and plan choice independently – metrics for plan quality independent of query optimization and query execution.

Session 2.2: Scalability

Session goal: identify risks to robustness, design tests specific to large deployments.

Scope: 1,000s of nodes, 1,000s of tables, 1,000s of users ...

Preparation: identify some issues, e.g., skew, concurrency control, workload management – prepare some presentation slide as introduction.

Discussion: forms of skew and imbalance – contributions of compile-time query optimization.

Session 2.3: Late binding

Session goal: identify & explain existing techniques, clarify the role of cost calculation, design tests.

Scope: run-time parameters, temporary tables, dynamic query execution plans, fluctuating system load & resource contention, user-defined functions, map-reduce, virtual & cloud deployments.

Preparation: draft a test plan for run-time parameters – prepare presentation slides as introduction.

Discussion: issues decidable or not during compile-time query optimization – required run-time techniques.

Session 2.4: Break out: test suite design

Session goal: design test suites, including specific SQL tables and queries, for cardinality estimation & cost calculation, plan generation & selection, robustness in high-scale deployments, dynamic query execution plans, plans for parallel execution, query plans with run-time parameters, specific tests for cloud deployments...

Topics: "vanilla" queries, updates & utilities, scalability, late binding. Preparation: paper skeletons.

Session 2.5: Paper preparation

Session goal: initialize one or more publications on robust query processing and measuring it (title, abstract, conclusions, outline).

4 Day 3: Query execution

Goals for the day:

- design regression tests to measure the contributions of query execution & access methods to robust query processing,
- identify concepts for possible publications.

Session 3.1: Robust execution algorithms

Session goal: list additional techniques, design tests for each of the techniques.
Scope: shared & coordinated scans, MDAM and scan versus probe, grow & shrink memory, skew bus- ter, shared intermediate results, sorting in index-to-index navigation, "bubblesort" for index join operations, dynamic query execution plans, eddies...
Preparation: prepare some introduction slides for each technique.
Discussion: scope & value & promise of each technique – how to measure & evaluate each technique – interaction of techniques, e.g., index-to-index navigation & bubblesort of index joins.

Session 3.2: Challenging deployments

Session goals:
- ensure a common understanding of existing techniques,
- identify possible metrics & tests.

Scope: poor estimation (cardinality estimation, cost calculation), scalability (skew, degree of parallelism), late binding (parameters, resources), mixed workloads (small & large, read-only & read-write).
Preparation: slides about performance effects of wrong join method, effects of skew, dynamic aggregation – metrics are the big open issue.
Discussion: what is required for a cloud deployment? – how can robustness in cloud execution be measured? – is robustness monitoring another cloud service?

Session 3.3: Excursion Walk in the woods ...

Session 3.4: Break out: test suite design

Session goal: design test suites for adaptive techniques and for their comparative evaluation, e.g., how valuable is sorting the join sequence if index nested loops join employs run generation & prefetch.
Preparation: paper skeletons.

Session 3.5: Paper preparation

Session goal: initialize one or more publications on robust query processing and measuring it (title, abstract, conclusions, outline).

4.1 Day 4: Physical database design

Goal for the day: design regression tests for logical and physical database design & required utilities.

Session 4.1: Physical database design

Session goals:

- scope physical database design & utilities,
- identify opportunities to improve robustness by appropriate physical database design,
- try to define metrics and tests.

Scope: Data placement (memory hierarchy), index tuning... driven by query optimization or query execution, required mechanisms (online, incremental...) – continuous tuning, adaptive indexing – limitations, stability, convergence, oscillations – are new index techniques required?

Preparation: introduction slides to physical database design & required database utilities.

Discussion: how can indexes improve robustness – compare local versus global indexes for robust query processing.

Session 4.2: Database utilities

Session goals:

- identify interference of utilities and query processing that lowers robustness,
- identify improvements to important database utilities,
- define metrics and tests.

Scope: index operations (e.g., creation), statistics update, defragmentation, merging (partitioned B-trees), roll-in & roll-out, backup and recovery, changes in logical database design, format changes in soft- ware updates, concurrency control on data & schema...

Preparation: slides to define online & incremental index operations, including "truly online" index operations based on multi-version concurrency control in the catalog tables.

Discussion: what utilities undermine robust query processing? – how to measure this interference.

Session 4.3: Adaptive index tuning

Session goal: ensure a common understanding of techniques and their potential & limitations.

Scope: hints from query optimization, external monitoring, adaptive indexing (database cracking, adaptive merging).

Preparation: prepare slides to explain techniques and differentiate their assumptions & values.

Discussion: index tuning by query optimization, query execution (hints embedded by query optimization), or access methods (side effect of index access) – capabilities & limitations of each approach – index removal heuristics – partial & incremental techniques (e.g., invalidation of key ranges).

Session 4.4: Break out: test suite design

Session goal: design test suites for database utilities and adaptive physical database design. Preparation: paper skeletons.

Session 4.5: Paper preparation

Session goal: initialize one or more publications on robust query processing and measuring it (title, abstract, conclusions, outline).

4.2 Day 5: System context

In parallel to Day 4; participants to choose among topics to create time for a break-out session. Goal for the day: identify relationships and opportunities – design regression tests for some of the adaptive mechanisms for some aspects of system context.

Session 5.1: Resource management

Session goals:
- ensure a common understanding & appreciation of resource shaping techniques,
- identify test requirements.

Scope: contention among already running queries (updates, utilities) – advantages & dangers of dynamic resource management.
Preparation: presentation slides to explain.
Discussion: How much testing is required before shipping query execution with dynamic resource management? How to minimize the test suite?

Session 5.2: Workload management

Session goal: clarify the interaction of workload management and robust query processing.
Scope: contention between running and waiting jobs, tasks, queries, updates – priorities, wait queues.
Preparation: presentation slides to explain.
Discussion: How can workload analysis assist in robust query processing? How can robustness be tested?

Session 5.3: Plan management

Session goal: compare techniques from Oracle, MS, Hewlett-Packard, etc.
Scope: plan caching, persistent plans, "hand" optimization, verification of plans, correction of plans and their cardinality estimation.
Preparation: presentation slides to explain techniques.
Discussion: what techniques are suitable for high scalability, mass deployments, cloud deployment, etc.?

Session 5.4: Configuration management

Session goal: Define regression tests to identify when a configuration change impacts the robustness advances & limitations of core performance factors (query optimization, query execution, physical database design). Initialize possible publications.
Scope: setup & patches, server options & best practices.
Preparation: Collect existing practices for specifying and validating configuration changes, horror sto- ries about configurations changes that impacted query processing robustness, how do existing configuration management tools enable correlation and root cause analysis?
Discussion: How to correlate configurations changes to measured results of testing plan management, workload management, and resource management robustness?

Session 5.5: Break out: test suite design

Session goal: design test suites for database utilities and adaptive physical database design.

Session 5.6: Paper preparation

Session goal: initialize one or more publications on robust query processing and measuring it (title, abstract, conclusions, outline).

5 Break-out sessions

Each day featured several breakout sessions, where the seminar was split in smaller groups. These sessions reflected each day's theme.

The goal of the first day's breakout sessions was test suite design, i.e. which aspects of robust query processing should be tested and which metrics are appropriate. The participants proposed several test suites, e.g. a benchmark inspired by tractor pulling in farming where the system is evaluated against an increasingly complex workload, as well as a benchmark for measuring end-to-end robustness of query processors considering two sources of performance variability: intrinsic variability reflecting the complexity of a query and extrinsic variability reflecting changes in the environment.

The second day's session were devoted to robustness issues in query optimization. The working groups discussed various aspects to assess and control the robustness of query optimization decisions by heuristic guidance and termination, the problem of cardinality estimation for queries with complex expressions and appropriate metrics, as well as reasons, metrics and benchmarks for risk in query optimization decisions.

Measuring the contribution of query execution and access methods to robust query processing was the goal of day 3. Specific topics of working groups in this area were the problem of deferring optimization decisions to query execution time by looking at concrete optimizer decisions and the spectra of interaction between optimization and execution as well as appropriate measures.

On the fourth day, two topics were addressed in parallel. In the first track of sessions, robustness of physical database design tools such as index selection tools and design advisors was discussed. The working groups proposed metrics and benchmarks allowing to evaluate robustness of these utilities. The second track of sessions addressed a broader system context by discussing the interaction of various system components in the relationship to robustness. A particular result of these working groups was a mixed workload benchmark bridging the gap between the existing single-workload suites and providing an opportunity to assess quality of workload management components in DBMS.

5.1 Definitions and tests

Tractor Pulling

Martin Kersten, Alfons Kemper, Volker Markl, Anisoara Nica, Meikel Poess, Kai-Uwe Sattler

Robustness of database systems under stress is hard to quantify, because there are many factors involved. Most notably the user expectation to perform a job within certain bounds of his expectation. The goal of this working group was to develop a database benchmark inspired by tractor pulling in farming.

In this benchmark, the tractor pull suite is formulated to evaluate a system systematically against an increasely complex workload in an automatic way. The parameter space for comparison of intra- and extra-solutions is defined with a metric based on the increasing variance in response time to enable relative comparisons of the solutions provided with a particular focus on robustness.

Black Hat Query Optimization

Guy Lohman, Rick Cole, Surajit Chaudhuri, Harumi Kuno

Categories of robustness testing (areas where "evil" queries are needed)

1. **Cardinality estimation** – sensitivity of cardinality estimation accuracy to
   - Correlation (Redundancy/too much information)
     - Correlation within tables
     - Correlation across tables (star schema skew across tables)
     - Multiple join predicates
   - Skew
2. **Plan generation** – Optimality of plans dependent on
   - accuracy of cardinality estimation
   - repertoire of possible plans

   – search space
3. **Execution-time** – Sensitivity to robust vs. non-robust execution operators

Problems in detecting / correcting:

– "Heisenbug" - when you stick probes in to measure some anomaly, it goes away.
– Sometimes “two wrongs can make a right”, i.e., off-setting errors, so that when one error is fixed, the other one appears.

Note: Need to balance accuracy of cardinality and cost models against cost (time, space) of attaining increased accuracy due to overhead of meta-data collection - e.g., actual execution counts, compile results, statistics collection - particularly at larger scales: more tables, more data, more predicates, and more complex queries.

Which offers more opportunity for disaster — robust query optimization or robust query execution? Guy observes (see war story, below) that cardinality estimation has the biggest impact, which far eclipses any other decision. This can affect all plan generation choices, esp. join order and join method. How to measure the next biggest opportunity after that? How to compare impact of fragile cardinality estimates vs. fragile operator implementations?

Guy’s cardinality estimation war story: Led lecture and practical example on "how to debug" for approx. 20 DBAs at large insurance company. Lead DBA provided practical exercise’s problem: EXPLAINs for two single-table queries consumed printouts 3/4 inches thick! Only difference was a single additional predicate on the second query, but the cardinality estimate varied by 7 orders of magnitude (database contained approx. 10M rows). Second query had one additional predicate that was a pseudo-key (first 4 bytes of last name + first and middle initial + zip code + last 4 bytes of Social Security Number. Since this was completely redundant of the information in the other predicates, it significantly underestimated the cardinality by 7 orders of magnitude (1 / # rows), causing a terrible plan, because of the underestimate.

There are three separate levels to measure:

1. The cardinality model: Input = a query and whatever statistics (meta-data) has been accumulated; Output = cardinality estimates.
2. The plan generation (and costing) model: Input = cardinality estimates and meta-data; Output = plan and its estimated cost.
3. The query execution engine: Input = plan and its estimated cost; Output = answers to query, actual cardinality and execution time (cost).

This results in three interesting tests of robustness:

1. Robustness of cardinality estimation accuracy.
2. Ability of query optimizer to produce a good (and robust) plan, given suspect cardinality estimates.
3. Robustness of a given plan, in terms of the executor’s ability to adapt it to run-time unknowns.

Measure (Time, Space, Accuracy)

How to measure robustness of cardinality accuracy?
Measure CPU cycles to do optimization (cost to create plan)
How to measure robustness of plan?
Execution changes

- Boundaries at which small changes make big differences in performance (buffer pool, sort heap)
- Hardware changes
- Switch workloads (varying ratio of updates) for a fixed schema
- Watching changes in behavior

We will focus on optimizer and cardinality errors. Specifically, one could measure estimated cardinality via EXPLAIN, and by adding LEO-like counters to runtime, measure the estimated percent of error in the cardinality estimate with | Estimate-Actual | / Actual

Benchmarking Robustness

Goetz Graefe, Jens Dittrich, Stefan Krompass, Thomas Neumann, Harald Schoening, Ken Salem

We investigate query processors with respect to their robustness against different ways of expressing a semantically equivalent query. Ideally, resources needed for execution should be identical, no matter how a query is stated. For example, SELECT 1 FROM A,B should behave the same as SELECT 1 FROM B,A. Various test sets for different aspects of a query have been defined, and a benchmark driver has been implemented, which allows to execute the test sets against different query processing engines. In addition to mere execution times, it may be worthwhile to also consider cardinality estimations (that should not only reasonably estimate the real result size but also should be in the same order of magnitude for semantically equivalent queries) and cost estimations. For the definition of test sets, we started with single-table queries (using equivalent expressions for simple arithmetics, for range queries, using negation etc.). As an example, a query using "…where not ($l_{shipdate}$! = '2002–01–13')" should be equivalent to a query "…where ($l_{shipdate}$ = '2002–01–13')", both in result and in behaviour. At first we did not assume access paths, then added simple and multi-column indexes. With respect to selection from multi-column indexes, restrictions might apply to leading, intermediate, or trailing index fields; they may be equality or range predicates; and there may be combinations. For example, an index on (A, B, C) should be used for "…where A = 4 and B between 7 and 11" as well as "…where A in $\{4, 7, 11\}$ and B between 7 and 11". As a further variable, we execute queries with literals versus queries with parameters. The various test sets will be executed against various open and commercial query engines.

Measuring end to end robustness for Query Processors

Parag Agrawal, Natassa Ailamaki, Nico Bruno, Leo Giakoumakis, Jayant Haritsa, Stratos Idreos, Wolfgang Lehner, Neoklis Polyzotis

We propose a benchmark for measuring end to end robustness of a query processor. Usually robustness of query processing is defined as consistency, predictability and the ability of the system to react smoothly in response to environment changes. We refine that definition by identifying two sources of performance variability: intrinsic variability which reflects the true complexity of the query in the new environment, and extrinsic variability, which stems from the inability of the system to model and adapt to changes in the environment. We believe that robustness should only measure extrinsic variability since any system would have to pay the cost of intrinsic variability to reflect the environment changes. To quantify this new notion of robustness we propose to measure the divergence (in execution time) between the plan produced by the query processor and the ideal plan for the corresponding environment. We couple that divergence metric with a performance metric and thus characterize robustness as the system's ability to minimize extrinsic variability. We complement that conceptual metric with various efficient and pragmatic algorithms for approximating the ideal plan, and environment changes that exercise known limitations of current query processors.

5.2 Query Optimization

Risk Reduction in Database Query Optimizers

Jayant Haritsa, Awny Al-Omari, Nicolas Bruno, Amol Deshpande, Leo Giakoumakis, Wey Guy, Alfons Kemper, Wolfgang Lehner, Alkis Polyzotis, Eric Simon

We have investigated and catalogued a variety of reasons due to which database query optimizers make errors in their selectivity estimations. The estimation errors have been transformed into a multi-dimensional space with each query having a compile-time and a run-time location in this space. The error metric is defined as the summation over the multi-dimensional space of the probability of each error times the adverse impact of that error. Several variants are feasible over this basic definition. A preliminary benchmark schema and a query have been identified for evaluating robustness of modern optimizers.

Robust Query Optimization: Cardinality estimation for queries with complex (known unknown) expressions

Anisoara Nica, Goetz Graefe, Nicolas Bruno, Rick Cole, Martin Kersten, Wolfgang Lehner, Guy Lohman, Robert Wrembel

A cost-based query optimizer of a relational database management system (RDBMS) uses cost estimates to compare enumerated query plans, hence relies on the accuracy of the cost model to find the best execution plan for a SQL statement. The cost model needs, among other metrics, two estimates to compute the cost of partial access plans: (1) the estimated size of the intermediate results which is based on the cardinality (i.e., the number of rows) estimations for the partial plans; and (2) the cost estimates of physical operators (e.g., hash joins) implemented in the execution engine of the RDBMS.

It is widely known that the cardinality estimation errors are often the cause of inefficient query plans generated by the query optimizers. The errors in cardinality estimations are largely due to (a) incorrect selectivity estimations of the individual predicates used in the SQL statement; (b) assumptions made about the correlations between predicates (e.g., independence assumptions); and (c) the semantic complexity of the query expressions (e.g., recursive query expressions).

Firstly, we classified the complex expressions ('known unknown') which can contribute to the cardinality estimation errors. The first category is related to selectivity estimation errors of individual predicates which includes predicates used in WHERE and ON clauses containing user defined functions (UDFs), built-in functions, subqueries, expressions on a derived table columns. The second category is related to the complexity of query constructs for which the cardinality estimation of the results is very hard to compute. This second category includes recursive query blocks, outer joins, grouped query blocks, distinct query blocks, nested query blocks, UNION, EXCEPT, and INTERSECT queries.

Secondly, we designed a benchmark which can be used to assess the robustness of the cardinality estimation of a query optimizer. The benchmark should include queries containing complex expressions classified above. The proposed metrics for assessing the robustness of a query optimizer are:

(I) For the best plan generated by the query optimizer, compute, for each physical operator, the cardinality estimation error.

Metric1 = SUM_all physical operators of the best plan
(| 'Estimated cardinality' - 'Actual cardinality' | / 'Actual cardinality').

'Actual cardinality' of a physical operator can be obtained by computing, in isolation, the result of the query subexpression corresponding to the subtree of that physical operator.

(II) Metric1 can be 0 (ideal) for the best plan. However, for many other plans which were enumerated by the query optimizer but pruned during the query optimization process Metric1 can be very large. Hence, a more realistic metric is:

Metric2 = SUM_all physical operators of the enumerated plans

(|'Estimated cardinality' - 'Actual cardinality' | / 'Actual cardinality').

(III) For each plan enumerated by the query optimizer, obtain its runtime by imposing that plan. 'RunTimeOpt' = minimum runtime among all enumerated access plans. 'RunTimeBest' = runtime of the best plan generated by the query optimizer

Metric3 = |RunTimeOpt - RunTimeBest| / RunTimeBest.

Last but not least, we discussed the new query landscape which contains generated queries. This new landscape includes very complex queries generated by new types of applications such as Object Relational Mapping (ORM) applications. The generated queries bring new types of problems to the query optimizer mainly due to the high degree of redundancy in the query semantics. As an example, the query Q1 below is generated by an Entity Framework application and it is semantically equivalent to the query Q2. However, many query optimizers may not have the highly sophisticated semantic transformations required to rewrite Q1 into Q2. Our take away message is that proposed techniques dealing with cardinality estimation computation for complex query expressions must be extended or redesigned to take into account this new complex query landscape.

```
Q1 = SELECT
[Project9].[ID] AS [ID],[Project9].[C1] AS [C1],[Project9].[C2] AS [C2],
[Project9].[ID1] AS [ID1],[Project9].[SalesOrderID] AS [SalesOrderID],
[Project9].[TotalDue] AS [TotalDue]
FROM
( SELECT [Distinct1].[ID] AS [ID], 1 AS [C1],
[Project8].[ID] AS [ID1], [Project8].[SalesOrderID] AS [SalesOrderID],
        [Project8].[TotalDue] AS [TotalDue], [Project8].[C1] AS [C2]
        FROM
(SELECT DISTINCT   [Extent1].[ID] AS [ID]
                 FROM  [DBA].[Person] AS [Extent1]
                                                 INNER JOIN [DBA].[Sales] AS [Extent2]
ON  EXISTS (SELECT   cast(1 as bit) AS [C1]
                        FROM    ( SELECT cast(1 as bit) AS X ) AS [SingleRowTable1]
                        LEFT OUTER JOIN  (SELECT [Extent3].[ID] AS [ID]
                                FROM [DBA].[Person] AS [Extent3]
WHERE [Extent2].[ID] = [Extent3].[ID] )AS [Project1] ON cast(1 as bit) = cast(1 as bit)
                        LEFT OUTER JOIN  (SELECT  [Extent4].[ID] AS [ID]
                                FROM [DBA].[Person] AS [Extent4]
WHERE [Extent2].[ID] = [Extent4].[ID] ) AS [Project2] ON cast(1 as bit) = cast(1 as bit)
                        WHERE ([Extent1].[ID] = [Project1].[ID]) OR (([Extent1].[ID] IS NULL)
AND ([Project2].[ID] IS NULL))  )
) AS [Distinct1]
  LEFT OUTER JOIN
 (SELECT  [Extent5].[ID] AS [ID],  [Extent6].[SalesOrderID] AS [SalesOrderID],  [Extent6].[TotalDue] AS [TotalDue],   1 AS [C1]
                FROM  [DBA].[Person] AS [Extent5]
                 INNER JOIN [DBA].[Sales] AS [Extent6]
   ON  EXISTS (SELECT    cast(1 as bit) AS [C1]
                        FROM    ( SELECT cast(1 as bit) AS X ) AS [SingleRowTable2]
                        LEFT OUTER JOIN  (SELECT  [Extent9].[ID] AS [ID]
                                FROM [DBA].[Person] AS [Extent9]
WHERE [Extent6].[ID] = [Extent9].[ID] )AS [Project5]
ON cast(1 as bit) = cast(1 as bit)
                        LEFT OUTER JOIN  (SELECT  [Extent8].[ID] AS [ID]
                                FROM [DBA].[Person] AS [Extent8]
WHERE [Extent6].[ID] = [Extent8].[ID] )
AS [Project6] ON cast(1 as bit) = cast(1 as bit)
                       WHERE ([Extent5].[ID] = [Project5].[ID])
OR (([Extent5].[ID] IS NULL) AND ([Project6].[ID] IS NULL))
                )
) AS [Project8]
ON ([Project8].[ID] = [Distinct1].[ID]) OR (([Project8].[ID] IS NULL)
AND ([Distinct1].[ID] IS NULL))
)  AS [Project9]
ORDER BY [Project9].[ID] ASC, [Project9].[C2] ASC
```

```
Q2 = select  Extent6.ID as,
1 as C1,
1 as C2,
Extent6.ID as ID1,
Extent6.SalesOrderID as SalesOrderID,
Extent6.TotalDue as TotalDue
 from DBA.Sales  as Extent6
order by Extent6.ID as ID asc
```

Consistent selectivity estimation via maximum entropy

Presented by Volker Markl. *Originally appeared in VLDB J. 16(1): 55-76 (2007)*
Authors : Volker Markl, Peter J. Haas, Marcel Kutsch, Nimrod Megiddo, Utkarsh Srivastava, Tam Minh Tran

To provide robust query performance, cost-based query optimizers need to estimate the selectivity of conjunctive predicates when comparing alternative query execution plans. To this end, advanced optimizers use multivariate statistics to improve information about the joint distribution of attribute values in a table. The joint distribution for all columns is almost always too large to store completely, and the resulting use of partial distribution information raises the possibility that multiple, non-equivalent selectivity estimates may be available for a given predicate. Current optimizers use cumbersome ad hoc methods to ensure that selectivities are estimated in a consistent manner. These methods ignore valuable information and tend to bias the optimizer toward query plans for which the least information is available, often yielding poor results. The maximum entropy method proposed by Markl et al. enables robust cardinality estimation based on the principle of maximum entropy (ME). The method exploits all available information and avoids the bias problem. In the absence of detailed knowledge, the ME approach reduces to standard uniformity and independence assumptions.

Measuring the Robustness of Query Optimization: Towards a Robustness Metric

Kai-Uwe Sattler, Meikel Poess, Florian Waas, Ken Salem, Harald Schoening, Glenn Paulley

We discussed the notions of robustness and performance and how to quantify them. The discussion focused on the family of simple parameterized range queries $Q = \{q_1, q_2, \ldots, q_n\}$, where the parameter controls query selectivity. For instance, $q_1 =$

```
SELECT count(*)
FROM T1
where T1.C1>=P1
  AND T1.C1<=P1;
```

We imagined a plot of query execution times as a function of selectivity, and we concluded that the performance could be measured as the difference between measured, $E(q_i))$, and optimal execution time, $O(q_i)$, such as: $P(q_i) = |O(q_i) - E(q_i)|$. It is unclear at this point how to obtain $(O(q_i)$. Robustness is related to the smoothness of the execution time function. We considered the coefficient of variation of the query performance of query set Q as a potential smoothness metric, i.e.

$$S(Q) = \frac{\sigma_Q}{\mu_Q} = \frac{\sqrt{\frac{1}{n} + \sum_{i=1}^{n} \left(P(q_i) - \frac{\sum_{i=1}^{n} P(q_i)}{n} \right)^2}}{\frac{\sum_{i=1}^{n} P(q_i)}{n}}.$$

Because performance and robustness are orthogonal metrics, it is necessary to consider both of them. While measurement of query execution time allows for the characterization of the performance and robustness of a Database Management System (DBMS), such measurements do not allow us to draw any conclusions about individual components of the DBMS, such as the query optimization. In a second approach we considered whether a comparison of measured and estimated query result cardinalities might be a good way to characterizing the performance and robustness. That is, for a set of queries we plans we extract the estimated top level cardinalities for each query: $C_e = \{e_1, e_2, \ldots, e_n\}$. Then, we compute the cardinalities by running the queries: $C_a = \{a_1, a_2, \ldots, a_n\}$. As the final metric, $C(Q)$, we considered the geometric mean of the error: $C(Q) = \sqrt[n]{\prod_{i=1}^{n} \frac{|a_i - e_i|}{a_i}}$.

Computing the cardinality estimate error is an imperfect measure of performance since estimation errors do not necessarily result in plan changes. It also fails to measure robustness. However, correlating measurements of cardinality estimation errors with execution time measurements might be a way of assuming responsibility of poor performance to the optimizer.

5.3 Query Execution

Deferring optimization decisions to query execution time using novel execution engine paradigms

Anastasia Ailamaki, Christian Koenig, Bernhard Mitschang, Martin Kersten, Stratos Idreos, Volker Markl, Eric Simon, Stefan Manegold, Goetz Graefe, Alfons Kemper, Anisoara Nica

When deciding on the least expensive plan for a query, the query optimizer uses two kinds of information:

1. Statistics on the data
2. Cost estimations for using the system's resources.

Both of the above are inaccurate. Statistics are invalidated by updates to the data and changes on value distributions, and real system costs can drift from estimations by orders of magnitude, for example in the face of competing workloads.

At execution time, we know more about data and the system than at optimization time. This raises an opportunity to reap the knowledge and "correct" optimizer's decisions. New query execution paradigms such as MonetDB and StagedDB act as enablers for such initiatives. MonetDB employs a "vertical" query processing optimization strategy which is based on architecture-conscious data partitioning and aggressive materializing of intermediate results for just-in-time optimal decisions. StagedDB optimizes sharing and parallelism by decoupling operators into independent services, and uses queues of requests to cohort-schedule queries with common subtrees in their plans. Both systems make it much easier to reconsider optimization decisions at processing time than a traditional query processing engine.

So which optimizer decisions can we reconsider? Clearly, join orders and selection of join algorithms play a key role in performance and can be heavily biased by inaccurate statistics towards a suboptimal plan. Join orders are mostly determined by cardinality estimations, where join algorithms are determined by other aspects of the data such as distribution and skew as well as sorting. As cardinality estimation is the Achilles' heel of most query optimizers, an effort to detect and reconsider an incorrect join order at query processing time.

A good example is the query "bubble", described as a join amongst several tables which is submitted to the query engine without a predetermined join order. In business processes (e.g. call center) the system accepts queries of the form Q(x1, x2, ..., xn) where xi is a variable; These can be either precompiled or they are compiled on the spot. Precompilation increases the opportunity for common subtrees, but ignores differences in optimal join orders because of changes in selectivity. Query-at-a-time compilation has the inverse effect, plus an additional optimization cost. The tradeoff is relaxed by sending a bubble describing a multi-way join to the query; the corresponding operator (e.g., m-join or eddies) execute the bubble, which then serves as a reusable block for other queries in the system. The problem with such operators is the increased memory requirements when many joins are executed on large datasets (e.g., multi-input hash join), or routing overhead (in the case of eddies).

We conclude that deferring or reconsidering optimization decisions to the query processor is a promising direction, however it is difficult to achieve predictably good results using traditional query processing engines. Novel query processing paradigms such as MonetDB and StagedDB strengthen the interaction between the database operators and the underlying system, perhaps offering potential to reconsider optimizer decisions without the associated processing overheads.

Testing how a query engine adapts to unexpected runtime environment

Eric Simon, Florian Waas, Bernhard Mitschang, Robert Wrembel

We first convened to define the robustness of a query processing engine as its capability (until certain limits) to adapt unexpected runtime environment. We considered a query plan generated by a query optimizer under certain assumptions on runtime environment. However, at the time a query plan is executed the assumptions may not hold, which may result in bad response time or throughput. We classified the parameters of the runtime environment into a category of query-related and query-unrelated parameters. The former category includes: data correlation between two columns, selectivity of filters with functions, selectivity of filters with arithmetic expressions. These parameters determine the size of the result of unary or binary operations. The second category includes: main memory, network speed and CPU. We focused our attention on main memory. Our simple idea of a robust test for this parameter is as follows. We consider a workload of queries, say TPC-H queries. We set up the memory parameter of the query engine with the amount of memory that is available on the system. For each query, we generate a query plan and we then execute the query using two sets of experiments. In the first set, we reduce the value of the static available memory for the query engine (by changing the memory parameter value). In the second set, we run an eager process concurrently with the query so that the available memory decreases as query execution goes. We measure response time in each case. We left open the implementation of a "portable test" for the second set of experiments. A similar testbed can then be constructed by running several instances of the same query (same query plan) in parallel. We then measure throughput.

A generalized join algorithm

Presented by Goetz Graefe, Hewlett-Packard Laboratories

Database query processing traditionally relies on three alternative join algorithms: index nested loops join exploits an index on its inner input, merge join exploits sorted inputs, and hash join exploits differences in the sizes of the join inputs. Cost-based query optimization chooses the most appropriate algorithm for each query and for each operation. Unfortunately, mistaken algorithm choices during compile-time query optimization are common yet expensive to investigate and to resolve.

Our goal is to end mistaken choices among join algorithms by replacing the three traditional join algorithms with a single one. Like merge join, this new join algorithm exploits sorted inputs. Like hash join, it exploits different input sizes for unsorted inputs. In fact, for unsorted inputs, the cost functions for recursive hash join and for hybrid hash join have guided our search for the new join algorithm. In consequence, the new join algorithm can replace both merge join and hash join in a database management system. The in-memory components of the new join algorithm employ indexes. If the database contains indexes for one (or both) of the inputs, the new join can exploit persistent indexes instead of temporary in-memory indexes. Using database indexes to match input records, the new join algorithm can also replace index nested loops join.

Interaction of Execution and Optimization

Surajit Chaudhuri, Jayant Haritsa, Stratos Idreos, Ihab Ilyas, Amol Deshpande, Rick Cole, Florian Waas, Awny Al-Omari, Guy Lohman

This breakout session focused on how to test the impact of interactions between the query optimizer and the execution engine on query processing robustness. The session began by considering whether robust query processing is equivalent to adaptive query processing. According to Meriam Webster, robust does not necessarily mean adaptive— robust means capable of operating under a wide range of conditions. According to that definition, in many ways, DBMSs are already robust. But what we are mean by query processing robustness is that systems should perform optimally under a wide range of conditions. This is much more stringent than robustness! Perhaps we should coin a new word ("roboptimal"? "adaptimal"? "optibust"? :) .

Performing optimally under a wide range of possibly unexpected conditions requires run-time adaptability. There could be several reasons we need to adapt a query plan. The plan could be perfect for expected conditions, but the execution environment could change at runtime, or the initial plan could have been too narrow due to incomplete knowledge, which we would need to mitigate at execution time even though the environment didn't change. Readers are referred to the definitive work by Amol Deshpande about adaptive query processing [Desphande 2007]. One can consider the problem of adaptation using an analogy to balancing an investment portfolio, where an aggressive investment paves the way to greater potential gain at the risk of greater loss. The question is how much effort to invest in making a plan adaptable, when to invest it, and how often and when (a priori? post mortem?) to re-balance the investment plan. The overhead, and the payoff, can be greater for parallel systems. For example, Teradata materializes result of each join for the sake of recoverability, but at the cost of increased IO. Some systems (Ingres, microstrategy, and map reduce.) also materialize intermediate results for the sake of safety, whereas other systems such as System R pipeline for the sake of performance.

There is a spectrum of runtime interaction for adaptive query optimization/processing. Dimensions include (1) how often adaptation takes place, (2) at what level the adjustment takes place, and (3) when the adaptation occurs. For (1), the spectrum runs the gamut from row-by-row (Eddies), to per plan segment (POP or Choose-Plan), per query, per session. With regard to (2), adaptation can take place at the macro-level (e.g. entire system) or the micro-level (operator parameter). As for (3), Choose-Plan adapts query plans a priori (to runtime). POP checks actual cardinalities against estimates and lazily/opportunistically adapts plans at runtime. LEO lets plans run to completion, then performs a post-mortem analysis for the benefit of future plans. (Note that POP and LEO are complementary — POP recognizes and avoids problems at runtime; LEO can then figure out the causes of problems.)

The final question is how to measure the impact of runtime interactions between the optimizer and the executor? Issues include what to measure (e.g., measure total resource usage?) and how to interpret it (how to distinguish whether a query is a victim or a perpetrator. Guy presented some slides showing how IBM demonstrated the impact of POP upon a customer workload.

Figure 1 shows aggregated improvement – the blue rectangles represent that the mid-50% of the queries in the workload had response times between 40 and 210 time units. The red lines above and below the rectangles represent the range of response times for the remaining outliers. This figure demonstrates that overall, although POP only modestly improved most queries' response times, it dramatically improved the response times of "problem" queries.

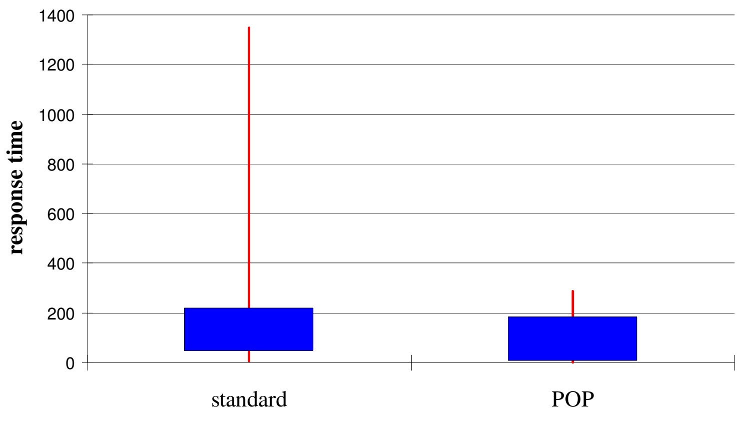

Fig. 1. Aggregated improvement.

Figure 2 is based on the same experiments as Figure 1, but shows speed up as a ratio of time without POP vs. time with POP. The queries are ordered by decreasing amount of improvement. The red line indicates the threshold of no speed up. This graph explicitly indicates regressions – queries that fall below the red line, whose speeds increased with POP, but does not intuitively convey the degree of either improvement or regression (e.g., for the query that regressed, how significant was the regression). Furthermore, this type of graph would not scale to show results for large numbers of queries.

Finally, Figure 3 is a scatter plot that shows both regression and improvement for each query. The X-axis represents response time without POP and the Y-axis represents response time with POP. The advantage of a scatter plot is that it gives both improvements and regressions the perspective of overall query response time. Also, a scatter plot scales naturally to show large numbers of data points.

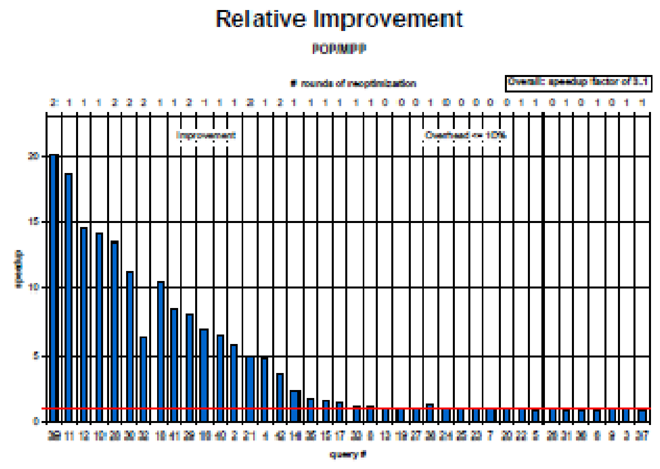

Fig. 2. Relative improvement with and without POP.

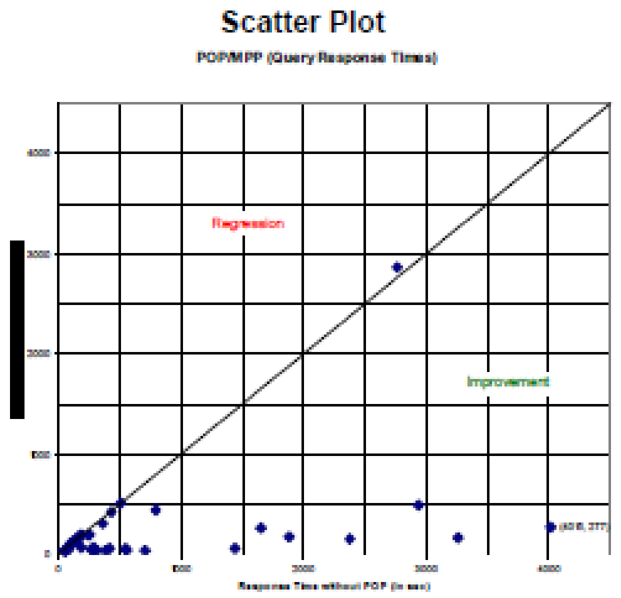

Fig. 3. Scatter plot.

5.4 Physical Database Designs and System Context

Heuristic Guidance and Termination of Query Optimization

Stefan Manegold, Anastassia Ailamaki, Stratos Idreos, Martin Kersten, Stefan Komprass, Guy Lohman, Thomas Neumann, Anisoara Nica

One aspect of robust query processing is the robustness of the query optimization process itself. We discussed various aspects to assess and control the robustness of query optimization decisions, focusing in particular on query parameters that are unknown at query compilation time. In addition, we discussed techniques to ensure the robustness of query optimization.

Evaluating the robustness of a physical database design advisor

Goetz Graefe, Anastassia Ailamaki, Stephan Ewen, Anisoara Nica, Robert Wrembel

A right physical database design (indexes, partitions, materialized views, and other data structures) strongly impacts query performance. Typically, a physical database design is done in two steps. In the first step, after designing a logical schema, the set of indexes, partitions, and materialized views is created by a DB designer based on the analysis of a DB schema and user requirements. In the second step, after deploying a database, the set of additional data structures is created (typically indexes and materialized views). This set is based on the analysis of a real workload and is very often proposed by a software, called physical database design advisor (PDBAdv). The disadvantage of the second step is that the set of data structures proposed by a PDBAdv optimizes the execution of a particular workload, assuming that it is static over time.

In order to evaluate the robustness of a PDBAdv and in order to be able to compare different PDBAdws we propose to use a standard TPC-H benchmark and a robustness evaluation method. The robustness evaluation method is as follows. First, the original TPC-H benchmark is run and for it, a PDBAdv proposes an initial physical DB design. For this design, the overall execution time (T0) is measured. Next, queries in TPC-H are being modified but retain their patterns. Different modifications constitute different workloads. For every modified workload W1, W2, ..., Wn, its overall execution time T1, T2, ..., Tn is measured for the initial DB design. The robustness of the PDBAdv is evaluated by the differences between T1 and T0, T2 and T0, etc. The maximum difference between the times is treated as a parameter.

Assessing the Robustness of Index Selection Tools

Stefan Manegold, Jens Dittrich, Stephan Ewen, Jayant Haritsa, Stratos Idreos, Wolfgang Lehner, Guy Lohman, Harald Schöning

Creating a suitable physical design, in particular creating the most beneficial indexes, is an important aspect of achieving high performance in database systems. Given a (historic) workload, automated index selection tools suggest a set of indexes to that should be created to achieve optimal performance for that workload, assuming that the given workload is representative for future workloads. In this work, we propose a metric to assess the robustness of such physical database design decisions wrt. varying future workloads. Focusing on single-table access, i.e., omitting joins for the time being, workloads differ in the in the set of columns they access in the select clause (projection list), where clause (local predicates), and group by clause. The idea is to create variations of a workload by varying these column sets and to compare both the estimated and real (measured) costs of the workload variation to the original workload given the physical design suggested for the original workload. The less to performance of the varied workloads on the original index set differs from the performance of the original workload, the more robust is the advice of the index selection tools.

Benchmarking Hybrid OLTP & OLAP Database Workloads

Alfons Kemper, Harumi Kuno, Glenn Paulley, Stefan Krompass, Eric Simon, Kai-Uwe Sattler, Leo Giakoumakis, Florian Waas, Rick Cole, Meikel Poess, Wey Guy, Ken Salem, Thomas Neumann

On Line Transaction Processing (OLTP) and Business Intelligence (BI) systems have performance characteristics inherent in their system design, database implementation and physical data layout. OLTP systems facilitate and manage transaction-oriented applications, which are vital to the day-to-day operations of a company. They are designed for data entry and retrieval transaction processing of relatively small amounts of data, e.g. row-wise lookup and update. Hence, OLTP systems require short latency, support for a high number of users and high availability. To the contrary, BI systems are usually deployed for the spotting, digging-out, and analyzing of large amounts of business data for the purpose of assisting with the decision making and planning of a company, e.g. computing sales revenue by products across regions and time, or fraud detection. Therefore, BI systems require fast scans, complex data algorithms for a relatively small number of users.

Because of their orthogonal performance characteristics, OLTP and BI databases have been kept and maintained on separate systems, coupled through Extract, Transform and Load (ETL) processes. However, due to companiesâ insatiable need for up-to-the-second business intelligence of their operational data, which, in many cases add Terabytes of data per day, recently the case has been made for operational or real-time Business Intelligence. The idea is to build systems that suit the performance requirements of both OLTP and BI systems. The advent of the first generation of such hybrid OLTP&BI systems requires means to characterize their performance. While there are standardized and widely used benchmarks addressing either OLTP or BI workloads, the lack of a hybrid benchmark led us to the definition of a new mixed workload benchmark, called TPC-CH. This new benchmark bridges the gap between the existing single-workload benchmarks: TPC-C for OLTP and TPC-H for BI.

The newly proposed TPC-CH benchmark executes a mixed workload: A transactional workload based on the order entry processing of TPC-C and a corresponding TPC-H equivalent BI query suite that operates on the same database. As it is derived from these two most widely used TPC benchmarks our new TPC-CH benchmark produces results that are highly comparable to both, hybrid systems and classic single-workload systems.

5.5 Test Suite Design

Measuring the Effects of Dynamic Activities in Data Warehouse Workloads on DBMS Performance

Leo Giakoumakis, Glenn Paulley, Meikel Poess, Ken Salem, Kai-Uwe Sattler, Robert Wrembel

Modern database management systems (DBMSs) are confronted with an array of dynamic runtime behaviors of data warehouse workloads. Users are connecting to systems at random times, issuing queries with vastly different workload characteristics. In order to guarantee high levels of overall system performance, DBMSs need to dynamically allocate the finite system resources to changing workloads.

In order to measure the ability of a DBMS to manage dynamic allocation of resources in response to a dynamic workload, we propose a benchmark. The benchmark is based on TPC-H. It focuses on testing systemsâĂŹ overall performance with respect to the management of main memory and the number of parallel processes. The benchmark is composed of two basic tests, namely a Fluctuating Memory Test (FMT) and a Fluctuating degree of Parallelism Test (FPT).

For both tests, baselines have to be defined. For FMT we define an upper baseline, further called memUBL, that represents the overall system's performance (an overall response time) for the whole test workload when the entire available memory is allocated to processing the workload. Similarly, we define a lower baseline, further called memLBL, that represents the overall system's performance when the minimum required memory was allocated for processing the workload. For FPT we also define two baselines in a similar manner. An upper baseline (procUBL) and a lower baseline (procLBL) represent the overall system's performance when all the available parallel processes and only one process were allocated to processing the workload, respectively. The procLBL and procUBL baselines can also be measured for single queries.

In FMT, the amount of available memory is dynamically changing when the test workload is executed. Memory can be decreased gradually from maximum to minimum, or can change randomly. During the execution of the test workload, an overall system's performance is measured. If the tested DBMS manages well the resources, then the observed performance characteristic oscillates between memUBL and memLBL.

In FPT, the amount of available parallel processes is decreased for each of the single queries from the TPC-H benchmark. For a query, say Qi, a tester defines the number of processes that should be allocated for executing the query. During the execution of Qi another query, say Qm, is executed that requires more processes than available. We measure how Qm impacts the performance of Qi. The number of processes required by Qm can be parameterized.

Finally, as a complex test we propose to merge FMT and FPT in order to represent the system's performance in a two-dimensional space.

6 Readling list

6.1 List

1. D. J. Abadi, S. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD Conference*, pages 967–980, 2008.
2. A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *SIGMOD Conference*, pages 181–192, 1999.
3. A. Aboulnaga and K. Salem. Report: 4th int'l workshop on self-managing database systems (smdb 2009). *IEEE Data Eng. Bull.*, 32(4):2–5, 2009.
4. P. Agarwal and B. Prabhakaran. Blind robust watermarking of 3d motion data. *TOMCCAP*, 6(1), 2010.
5. S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, pages 496–505, 2000.
6. M. Ahmad, A. Aboulnaga, and S. Babu. Query interactions in database workloads. In *DBTest*, 2009.
7. S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. L. Perez. The datapath system: a data-centric analytic processing engine for large data warehouses. In *SIGMOD Conference*, pages 519–530, 2010.
8. A. Avanes and J. C. Freytag. Adaptive workflow scheduling under resource allocation constraints and network dynamics. *PVLDB*, 1(2):1631–1637, 2008.
9. R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD Conference*, pages 261–272, 2000.
10. B. Babcock and S. Chaudhuri. Towards a robust query optimizer: A principled and practical approach. In *SIGMOD Conference*, pages 119–130, 2005.
11. S. Babu and P. Bizarro. Adaptive query processing in the looking glass. In *CIDR*, pages 238–249, 2005.
12. S. Babu, P. Bizarro, and D. J. DeWitt. Proactive re-optimization. In *SIGMOD Conference*, pages 107–118, 2005.

13. S. Babu, N. Borisov, S. Uttamchandani, R. Routray, and A. Singh. Diads: Addressing the "my-problem-or-yours" syndrome with integrated san and database diagnosis. In *FAST*, pages 57–70, 2009.
14. A. Behm, V. Markl, P. J. Haas, and K. Murthy. Integrating query-feedback based statistics into informix dynamic server. In *BTW*, pages 582–601, 2007.
15. P. Bizarro, S. Babu, D. J. DeWitt, and J. Widom. Content-based routing: Different plans for different data. In *VLDB*, pages 757–768, 2005.
16. P. Bizarro, N. Bruno, and D. J. DeWitt. Progressive parametric query optimization. *IEEE Trans. Knowl. Data Eng.*, 21(4):582–594, 2009.
17. M. Böhm, D. Habich, W. Lehner, and U. Wloka. Dipbench toolsuite: A framework for benchmarking integration systems. In *ICDE*, pages 1596–1599, 2008.
18. P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, 2008.
19. P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. *PVLDB*, 2(2):1648–1653, 2009.
20. N. Borisov, S. Babu, S. Uttamchandani, R. Routray, and A. Singh. Diads: A problem diagnosis tool for databases and storage area networks. *PVLDB*, 2(2):1546–1549, 2009.
21. N. Borisov, S. Babu, S. Uttamchandani, R. Routray, and A. Singh. Why did my query slow down? *CoRR*, abs/0907.3183, 2009.
22. I. T. Bowman, P. Bumbulis, D. Farrar, A. K. Goel, B. Lucier, A. Nica, G. N. Paulley, J. Smirnios, and M. Young-Lai. Sql anywhere: A holistic approach to database self-management. In *ICDE Workshops*, pages 414–423, 2007.
23. I. T. Bowman and G. N. Paulley. Join enumeration in a memory-constrained environment. In *ICDE*, pages 645–654, 2000.
24. N. Bruno. Minimizing database repros using language grammars. In *EDBT*, pages 382–393, 2010.
25. N. Bruno and P. Castro. Towards declarative queries on adaptive data structures. In *ICDE*, pages 1249–1258, 2008.
26. N. Bruno and S. Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *SIGMOD Conference*, pages 227–238, 2005.
27. N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *ICDE*, pages 826–835, 2007.
28. N. Bruno and S. Chaudhuri. Online autoadmin: (physical design tuning). In *SIGMOD Conference*, pages 1067–1069, 2007.
29. N. Bruno and S. Chaudhuri. Interactive physical design tuning. In *ICDE*, pages 1161–1164, 2010.
30. N. Bruno, S. Chaudhuri, and R. Ramamurthy. Power hints for query optimization. In *ICDE*, pages 469–480, 2009.
31. N. Bruno, S. Chaudhuri, and G. Weikum. Database tuning using online algorithms. In *Encyclopedia of Database Systems*, pages 741–744. Springer, 2009.
32. N. Bruno and R. V. Nehme. Configuration-parametric query optimization for physical design tuning. In *SIGMOD Conference*, pages 941–952, 2008.
33. N. Bruno and R. V. Nehme. Finding min-repros in database software. In *DBTest*, 2009.
34. N. Bruno and R. V. Nehme. Mini-me: A min-repro system for database software. In *ICDE*, pages 1153–1156, 2010.
35. G. Candea, N. Polyzotis, and R. Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. *PVLDB*, 2(1):277–288, 2009.

36. S. Chaudhuri, V. Ganti, and R. Motwani. Robust identification of fuzzy duplicates. In *ICDE*, pages 865–876, 2005.
37. S. Chaudhuri, L. Giakoumakis, V. R. Narasayya, and R. Ramamurthy. Rule profiling for query optimizers and their implications. In *ICDE*, pages 1072–1080, 2010.
38. S. Chaudhuri, R. Kaushik, and R. Ramamurthy. When can we trust progress estimators for sql queries? In *SIGMOD Conference*, pages 575–586, 2005.
39. S. Chaudhuri, A. C. König, and V. R. Narasayya. Sqlcm: A continuous monitoring framework for relational database engines. In *ICDE*, pages 473–485, 2004.
40. S. Chaudhuri and V. R. Narasayya. Automating statistics management for query optimizers. In *ICDE*, pages 339–348, 2000.
41. S. Chaudhuri, V. R. Narasayya, and R. Ramamurthy. A pay-as-you-go framework for query execution feedback. *PVLDB*, 1(1):1141–1152, 2008.
42. S. Chaudhuri and G. Weikum. Rethinking database system architecture: Towards a self-tuning risc-style database system. In *VLDB*, pages 1–10, 2000.
43. S. Chaudhuri and G. Weikum. Foundations of automated database tuning. In *ICDE*, page 104, 2006.
44. S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Inspector joins. In *VLDB*, pages 817–828, 2005.
45. Y. Chen, R. L. Cole, W. J. McKenna, S. Perfilov, A. Sinha, and E. S. Jr. Partial join order optimization in the paraccel analytic database. In *SIGMOD Conference*, pages 905–908, 2009.
46. F. C. Chu, J. Y. Halpern, and P. Seshadri. Least expected cost query optimization: An exercise in utility. In *PODS*, pages 138–147, 1999.
47. R. L. Cole. A decision theoretic cost model for dynamic plans. *IEEE Data Eng. Bull.*, 23(2):34–41, 2000.
48. R. L. Cole and G. Graefe. Optimization of dynamic query evaluation plans. In *SIGMOD Conference*, pages 150–160, 1994.
49. G. C. Das and J. R. Haritsa. Robust heuristics for scalable optimization of complex sql queries. In *ICDE*, pages 1281–1283, 2007.
50. U. Dayal, H. A. Kuno, J. L. Wiener, K. Wilkinson, A. Ganapathi, and S. Krompass. Managing operational business intelligence workloads. *Operating Systems Review*, 43(1):92–98, 2009.
51. A. Deshpande. Adaptive query processing with eddies. In *IIT Bombay tutorial*, 2006.
52. A. Deshpande, Z. G. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.
53. A. Dey, S. Bhaumik, Harish D., and J. R. Haritsa. Efficiently approximating query optimizer plan diagrams. *PVLDB*, 1(2):1325–1336, 2008.
54. N. Dieu, A. Dragusanu, F. Fabret, F. Llirbat, and E. Simon. 1, 000 tables inside the from. *PVLDB*, 2(2):1450–1461, 2009.
55. J. Dittrich, L. Blunschi, and M. A. V. Salles. Dwarfs in the rearview mirror: how big are they really? *PVLDB*, 1(2):1586–1597, 2008.
56. J.-P. Dittrich, P. M. Fischer, and D. Kossmann. Agile: Adaptive indexing for context-aware information filters. In *SIGMOD Conference*, pages 215–226, 2005.
57. S. Duan, S. Babu, and K. Munagala. Fa: A system for automating failure diagnosis. In *ICDE*, pages 1012–1023, 2009.
58. A. El-Helw, I. F. Ilyas, W. Lau, V. Markl, and C. Zuzarte. Collecting and maintaining just-in-time statistics. In *ICDE*, pages 516–525, 2007.

59. A. K. Elmagarmid and D. Agrawal, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. ACM, 2010.
60. S. Ewen, H. Kache, V. Markl, and V. Raman. Progressive query optimization for federated queries. In *EDBT*, pages 847–864, 2006.
61. S. Ewen, M. Ortega-Binderberger, and V. Markl. A learning optimizer for a federated database management system. *Inform., Forsch. Entwickl.*, 20(3):138–151, 2005.
62. T. Fiebig and H. Schöning. Software ag's tamino xquery processor. In *XIME-P*, pages 19–24, 2004.
63. D. Florescu and D. Kossmann. Rethinking cost and performance of database systems. *SIGMOD Record*, 38(1):43–48, 2009.
64. M. J. Franklin, S. Krishnamurthy, N. Conway, A. Li, A. Russakovsky, and N. Thombre. Continuous analytics: Rethinking query processing in a network-effect world. In *CIDR*, 2009.
65. P. W. Frey, R. Goncalves, M. L. Kersten, and J. Teubner. Spinning relations: high-speed networks for distributed join processing. In *DaMoN*, pages 27–33, 2009.
66. C. A. Galindo-Legaria, T. Grabs, C. Kleinerman, and F. Waas. Database change notifications: Primitives for efficient database query result caching. In *VLDB*, pages 1275–1278, 2005.
67. K. E. Gebaly and A. Aboulnaga. Robustness in automatic physical database design. In *EDBT*, pages 145–156, 2008.
68. A. Ghazal, A. Crolotte, and D. Y. Seid. Recursive sql query optimization with k-iteration lookahead. In *DEXA*, pages 348–357, 2006.
69. A. Ghazal, D. Y. Seid, R. Bhashyam, A. Crolotte, M. Koppuravuri, and V. G. Dynamic plan generation for parameterized queries. In *SIGMOD Conference*, pages 909–916, 2009.
70. A. Ghazal, D. Y. Seid, A. Crolotte, and B. McKenna. Exploiting interactions among query rewrite rules in the teradata dbms. In *DEXA*, pages 596–609, 2008.
71. D. Gmach, S. Krompass, A. Scholz, M. Wimmer, and A. Kemper. Adaptive quality of service management for enterprise services. *TWEB*, 2(1), 2008.
72. D. Gmach, S. Krompass, S. Seltzsam, M. Wimmer, and A. Kemper. Dynamic load balancing of virtualized database services using hints and load forecasting. In *CAiSE Workshops (2)*, pages 23–37, 2005.
73. D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Capacity management and demand prediction for next generation data centers. In *ICWS*, pages 43–50, 2007.
74. G. Graefe. The value of merge-join and hash-join in sql server. In *VLDB*, pages 250–253, 1999.
75. G. Graefe. Dynamic query evaluation plans: Some course corrections? *IEEE Data Eng. Bull.*, 23(2):3–6, 2000.
76. G. Graefe. Executing nested queries. In *BTW*, pages 58–77, 2003.
77. G. Graefe and H. A. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*, pages 371–381, 2010.
78. G. Graefe, H. A. Kuno, and J. L. Wiener. Visualizing the robustness of query execution. In *CIDR*, 2009.
79. P. J. Haas, I. F. Ilyas, G. M. Lohman, and V. Markl. Discovering and exploiting statistical properties for query optimization in relational databases: A survey. *Statistical Analysis and Data Mining*, 1(4):223–250, 2009.
80. W.-S. Han, W. Kwak, J. Lee, G. M. Lohman, and V. Markl. Parallelizing query optimization. *PVLDB*, 1(1):188–200, 2008.

81. W.-S. Han and J. Lee. Dependency-aware reordering for parallelizing query optimization in multi-core cpus. In *SIGMOD Conference*, pages 45–58, 2009.
82. W.-S. Han, J. Ng, V. Markl, H. Kache, and M. Kandil. Progressive optimization in a shared-nothing parallel database. In *SIGMOD Conference*, pages 809–820, 2007.
83. Harish D., P. N. Darera, and J. R. Haritsa. On the production of anorexic plan diagrams. In *VLDB*, pages 1081–1092, 2007.
84. Harish D., P. N. Darera, and J. R. Haritsa. Identifying robust plans through plan diagram reduction. *PVLDB*, 1(1):1124–1140, 2008.
85. S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In *SIGMOD Conference*, pages 383–394, 2005.
86. S. Helmer, T. Neumann, and G. Moerkotte. Estimating the output cardinality of partial preaggregation with a measure of clusteredness. In *VLDB*, pages 656–667, 2003.
87. S. Helmer, T. Neumann, and G. Moerkotte. A robust scheme for multilevel extendible hashing. In *ISCIS*, pages 220–227, 2003.
88. H. Herodotou and S. Babu. Automated sql tuning through trial and (sometimes) error. In *DBTest*, 2009.
89. M. Holze and N. Ritter. Towards workload shift detection and prediction for autonomic databases. In *PIKM*, pages 109–116, 2007.
90. M. Holze and N. Ritter. System models for goal-driven self-management in autonomic databases. In *KES (2)*, pages 82–90, 2009.
91. M. Hong, M. Riedewald, C. Koch, J. Gehrke, and A. J. Demers. Rule-based multi-query optimization. In *EDBT*, pages 120–131, 2009.
92. K. Hose, D. Klan, and K.-U. Sattler. Online tuning of aggregation tables for olap. In *ICDE*, pages 1679–1686, 2009.
93. S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, pages 68–78, 2007.
94. S. Idreos, M. L. Kersten, and S. Manegold. Updating a cracked database. In *SIGMOD Conference*, pages 413–424, 2007.
95. S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *SIGMOD Conference*, pages 297–308, 2009.
96. I. F. Ilyas, W. G. Aref, A. K. Elmagarmid, H. G. Elmongui, R. Shah, and J. S. Vitter. Adaptive rank-aware query optimization in relational databases. *ACM Trans. Database Syst.*, 31(4):1257–1304, 2006.
97. I. F. Ilyas, V. Markl, P. J. Haas, P. Brown, and A. Aboulnaga. Cords: Automatic discovery of correlations and soft functional dependencies. In *SIGMOD Conference*, pages 647–658, 2004.
98. M. Ivanova, M. L. Kersten, and N. Nes. Adaptive segmentation for scientific databases. In *ICDE*, pages 1412–1414, 2008.
99. M. Ivanova, M. L. Kersten, and N. Nes. Self-organizing strategies for a column-store database. In *EDBT*, pages 157–168, 2008.
100. Z. G. Ives, A. Deshpande, and V. Raman. Adaptive query processing: Why, how, when, and what next? In *VLDB*, pages 1426–1427, 2007.
101. R. Johnson, N. Hardavellas, I. Pandis, N. Mancheril, S. Harizopoulos, K. Sabirli, A. Ailamaki, and B. Falsafi. To share or not to share? In *VLDB*, pages 351–362, 2007.
102. R. Johnson, I. Pandis, and A. Ailamaki. Critical sections: re-emerging scalability concerns for database storage engines. In *DaMoN*, pages 35–40, 2008.
103. R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-mt: a scalable storage manager for the multicore era. In *EDBT*, pages 24–35, 2009.

104. R. Johnson, R. Stoica, A. Ailamaki, and T. C. Mowry. Decoupling contention management from scheduling. In *ASPLOS*, pages 117–128, 2010.
105. B. T. Jónsson, M. Arinbjarnar, B. THórsson, M. J. Franklin, and D. Srivastava. Performance and overhead of semantic cache management. *ACM Trans. Internet Techn.*, 6(3):302–331, 2006.
106. N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD Conference*, pages 106–117, 1998.
107. H. Kache, W.-S. Han, V. Markl, V. Raman, and S. Ewen. Pop/fed: Progressive query optimization for federated queries in db2. In *VLDB*, pages 1175–1178, 2006.
108. R. A. Kader, P. A. Boncz, S. Manegold, and M. van Keulen. Rox: run-time optimization of xqueries. In *SIGMOD Conference*, pages 615–626, 2009.
109. R. A. Kader, P. A. Boncz, S. Manegold, and M. van Keulen. Rox: The robustness of a run-time xquery optimizer against correlated data. In *ICDE*, pages 1185–1188, 2010.
110. M. L. Kersten. Database architecture fertilizers: Just-in-time, just-enough, and autonomous growth. In *EDBT*, page 1, 2006.
111. M. L. Kersten and S. Manegold. Cracking the database store. In *CIDR*, pages 213–224, 2005.
112. A. C. König and S. U. Nabar. Scalable exploration of physical database design. In *ICDE*, page 37, 2006.
113. A. C. König and G. Weikum. Combining histograms and parametric curve fitting for feedback-driven query result-size estimation. In *VLDB*, pages 423–434, 1999.
114. A. C. König and G. Weikum. A framework for the physical design problem for data synopses. In *EDBT*, pages 627–645, 2002.
115. A. C. König and G. Weikum. Automatic tuning of data synopses. *Inf. Syst.*, 28(1-2):85–109, 2003.
116. S. Krompass, D. Gmach, A. Scholz, S. Seltzsam, and A. Kemper. Quality of service enabled database applications. In *ICSOC*, pages 215–226, 2006.
117. S. Krompass, H. A. Kuno, J. L. Wiener, K. Wilkinson, U. Dayal, and A. Kemper. Managing long-running queries. In *EDBT*, pages 132–143, 2009.
118. S. Krompass, H. A. Kuno, J. L. Wiener, K. Wilkinson, U. Dayal, and A. Kemper. A testbed for managing dynamic mixed workloads. *PVLDB*, 2(2):1562–1565, 2009.
119. H. A. Kuno, U. Dayal, J. L. Wiener, K. Wilkinson, A. Ganapathi, and S. Krompass. Managing dynamic mixed workloads for operational business intelligence. In *DNIS*, pages 11–26, 2010.
120. P.-Å. Larson, W. Lehner, J. Zhou, and P. Zabback. Exploiting self-monitoring sample views for cardinality estimation. In *SIGMOD Conference*, pages 1073–1075, 2007.
121. A. J. Lee, A. Koeller, A. Nica, and E. A. Rundensteiner. Data warehouse evolution: Trade-offs between quality and cost of query rewritings. In *ICDE*, page 255, 1999.
122. A. W. Lee and M. Zaït. Closing the query processing loop in oracle 11g. *PVLDB*, 1(2):1368–1378, 2008.
123. A. W. Lee, M. Zaït, T. Cruanes, R. Ahmed, and Y. Zhu. Validating the oracle sql engine. In *DBTest*, 2009.
124. W. Lehner. Robust data management. In *Mobile Information Management*, 2004.
125. Q. Li, M. Shao, V. Markl, K. S. Beyer, L. S. Colby, and G. M. Lohman. Adaptively reordering joins during query execution. In *ICDE*, pages 26–35, 2007.
126. M. Lühring, K.-U. Sattler, K. S. 0002, and E. Schallehn. Autonomous management of soft indexes. In *ICDE Workshops*, pages 450–458, 2007.

127. G. Luo, J. F. Naughton, C. J. Ellmann, and M. Watzke. Toward a progress indicator for database queries. In *SIGMOD Conference*, pages 791–802, 2004.
128. G. Luo, J. F. Naughton, C. J. Ellmann, and M. Watzke. Increasing the accuracy and coverage of sql progress indicators. In *ICDE*, pages 853–864, 2005.
129. G. Luo, J. F. Naughton, C. J. Ellmann, and M. Watzke. Transaction reordering with application to synchronized scans. In *CIKM*, pages 1335–1336, 2008.
130. G. Luo, J. F. Naughton, and P. S. Yu. Multi-query sql progress indicators. In *EDBT*, pages 921–941, 2006.
131. G. Luo, C. Tang, and P. S. Yu. Resource-adaptive real-time new event detection. In *SIGMOD Conference*, pages 497–508, 2007.
132. S. Manegold, P. A. Boncz, and M. L. Kersten. What happens during a join? dissecting cpu and memory optimization effects. In *VLDB*, pages 339–350, 2000.
133. S. Manegold and I. Manolescu. Performance evaluation in database research: principles and experience. In *EDBT*, page 1156, 2009.
134. S. Manegold, A. Pellenkoft, and M. L. Kersten. A multi-query optimizer for monet. In *BNCOD*, pages 36–50, 2000.
135. S. Manegold, F. Waas, and D. Gudlat. In quest of the bottleneck - monitoring parallel database systems. In *PVM/MPI*, pages 277–284, 1997.
136. I. Manolescu and S. Manegold. Performance evaluation and experimental assessment - conscience or curse of database research? In *VLDB*, pages 1441–1442, 2007.
137. A. Marian, S. Amer-Yahia, N. Koudas, and D. Srivastava. Adaptive processing of top-k queries in xml. In *ICDE*, pages 162–173, 2005.
138. V. Markl. Mistral - processing relational queries using a multidimensional access technique. *Datenbank Rundbrief*, 26:24–25, 2000.
139. V. Markl, P. J. Haas, M. Kutsch, N. Megiddo, U. Srivastava, and T. M. Tran. Consistent selectivity estimation via maximum entropy. *VLDB J.*, 16(1):55–76, 2007.
140. V. Markl, G. M. Lohman, and V. Raman. Leo: An autonomic query optimizer for db2. *IBM Systems Journal*, 42(1):98–106, 2003.
141. V. Markl, V. Raman, D. E. Simmen, G. M. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In *SIGMOD Conference*, pages 659–670, 2004.
142. C. Mishra and N. Koudas. The design of a query monitoring system. *ACM Trans. Database Syst.*, 34(1), 2009.
143. G. Moerkotte, T. Neumann, and G. Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *PVLDB*, 2(1):982–993, 2009.
144. T. Neumann. Query simplification: graceful degradation for join-order optimization. In *SIGMOD Conference*, pages 403–414, 2009.
145. T. Neumann and G. Moerkotte. A framework for reasoning about share equivalence and its integration into a plan generator. In *BTW*, pages 7–26, 2009.
146. T. Neumann and G. Weikum. Scalable join processing on very large rdf graphs. In *SIGMOD Conference*, pages 627–640, 2009.
147. A. Nica, D. S. Brotherston, and D. W. Hillis. Extreme visualisation of query optimizer search space. In *SIGMOD Conference*, pages 1067–1070, 2009.
148. O. Ozmen, K. Salem, J. Schindler, and S. Daniel. Workload-aware storage layout for database systems. In *SIGMOD Conference*, pages 939–950, 2010.
149. G. N. Paulley and W. B. Cowan. A conceptual framework for error analysis in sql interfaces. In *IDS*, pages 406–430, 1992.

150. A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD Conference*, pages 165–178, 2009.
151. N. Polyzotis and Y. E. Ioannidis. Speculative query processing. In *CIDR*, 2003.
152. M. Pöss, R. O. Nambiar, and D. Walrath. Why you should run tpc-ds: A workload analysis. In *VLDB*, pages 1138–1149, 2007.
153. V. Raman, A. Deshpande, and J. M. Hellerstein. Using state modules for adaptive query processing. In *ICDE*, pages 353–, 2003.
154. V. Raman, V. Markl, D. E. Simmen, G. M. Lohman, and H. Pirahesh. Progressive optimization in action. In *VLDB*, pages 1337–1340, 2004.
155. V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. pages 60–69, 2008.
156. N. Reddy and J. R. Haritsa. Analyzing plan diagrams of database query optimizers. In *VLDB*, pages 1228–1240, 2005.
157. N. Ritter, B. Mitschang, T. Härder, M. Gesmann, and H. Schöning. Capturing design dynamics the concord approach. In *ICDE*, pages 440–451, 1994.
158. F. Rosenthal, P. B. Volk, M. Hahmann, D. Habich, and W. Lehner. Drift-aware ensemble regression. In *MLDM*, pages 221–235, 2009.
159. K.-U. Sattler, I. Geist, and E. Schallehn. Quiet: Continuous query-driven index tuning. In *VLDB*, pages 1129–1132, 2003.
160. K.-U. Sattler and W. Lehner. Quality of service and predictability in dbms. In *EDBT*, page 748, 2008.
161. K.-U. Sattler, E. Schallehn, and I. Geist. Autonomous query-driven index tuning. In *IDEAS*, pages 439–448, 2004.
162. K.-U. Sattler, E. Schallehn, and I. Geist. Towards indexing schemes for self-tuning dbms. In *ICDE Workshops*, page 1216, 2005.
163. A. Schmidt, F. Waas, M. L. Kersten, D. Florescu, M. J. Carey, I. Manolescu, and R. Busse. Why and how to benchmark xml databases. *SIGMOD Record*, 30(3):27–32, 2001.
164. S. Schmidt, T. Legler, S. Schär, and W. Lehner. Robust real-time query processing with qstream. In *VLDB*, pages 1299–1302, 2005.
165. K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. Colt: continuous on-line tuning. In *SIGMOD Conference*, pages 793–795, 2006.
166. K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. On-line index selection for shifting workloads. In *ICDE Workshops*, pages 459–468, 2007.
167. K. Schnaitter and N. Polyzotis. A benchmark for online index selection. In *ICDE*, pages 1701–1708, 2009.
168. K. Schnaitter and N. Polyzotis. Semi-automatic index tuning: Keeping dbas in the loop. *CoRR*, abs/1004.1249, 2010.
169. K. Schnaitter, N. Polyzotis, and L. Getoor. Index interactions in physical design tuning: Modeling, analysis, and applications. *PVLDB*, 2(1):1234–1245, 2009.
170. H. Schöning. The adabas buffer pool manager. In A. Gupta, O. Shmueli, and J. Widom, editors, *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 675–679. Morgan Kaufmann, 1998.
171. K. Seidler, E. Peukert, G. Hackenbroich, and W. Lehner. Approximate query answering and result refinement on xml data. In *SSDBM*, pages 78–86, 2010.
172. M. Shao, S. W. Schlosser, S. Papadomanolakis, J. Schindler, A. Ailamaki, and G. R. Ganger. Multimap: Preserving disk locality for multidimensional datasets. In *ICDE*, pages 926–935, 2007.

173. P. Shivam, V. Marupadi, J. S. Chase, T. Subramaniam, and S. Babu. Cutting corners: Workbench automation for server benchmarking. In *USENIX Annual Technical Conference*, pages 241–254, 2008.
174. M. Sinnwell and A. C. König. Managing distributed memory to meet multiclass workload response time goals. In *ICDE*, pages 87–94, 1999.
175. A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. *ACM Trans. Database Syst.*, 35(1), 2010.
176. M. Spiliopoulou and J. C. Freytag. Modelling resource utilization in pipelined query execution. In *Euro-Par, Vol. I*, pages 872–880, 1996.
177. B. Stegmaier and R. Kuntschke. Streamglobe: Adaptive anfragebearbeitung und optimierung auf datenströmen. In *GI Jahrestagung (1)*, pages 367–372, 2004.
178. M. Stillger and J. C. Freytag. Testing the quality of a query optimizer. *IEEE Data Eng. Bull.*, 18(3):41–48, 1995.
179. M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. Leo - db2's learning optimizer. In *VLDB*, pages 19–28, 2001.
180. M. Thiele, A. Bader, and W. Lehner. Multi-objective scheduling for real-time data warehouses. In *BTW*, pages 307–326, 2009.
181. A. Thiem and K.-U. Sattler. An integrated approach to performance monitoring for autonomous tuning. In *ICDE*, pages 1671–1678, 2009.
182. S. Tozer, T. Brecht, and A. Aboulnaga. Q-cop: Avoiding bad query mixes to minimize client timeouts under heavy loads. In *ICDE*, pages 397–408, 2010.
183. O. Trajman, A. Crolotte, D. Steinhoff, R. O. Nambiar, and M. Poess. Database are not toasters: A framework for comparing data warehouse appliances. In *TPCTC*, pages 31–51, 2009.
184. P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. *PVLDB*, 2(1):706–717, 2009.
185. M. Vieira and H. Madeira. A dependability benchmark for oltp application environments. In *VLDB*, pages 742–753, 2003.
186. G. von Bochmann, A. Das, R. Dssouli, M. Dubuc, A. Ghedamsi, and G. Luo. Fault models in testing. In *Protocol Test Systems*, pages 17–30, 1991.
187. K. Vorwerk and G. N. Paulley. On implicate discovery and query optimization. In *IDEAS*, pages 2–11, 2002.
188. F. Waas and A. Pellenkoft. Join order selection - good enough is easy. In *BNCOD*, pages 51–67, 2000.
189. F. M. Waas. Beyond conventional data warehousing - massively parallel data processing with greenplum database. In *BIRTE (Informal Proceedings)*, 2008.
190. F. M. Waas and J. M. Hellerstein. Parallelizing extensible query optimizers. In *SIGMOD Conference*, pages 871–878, 2009.
191. G. Weikum, A. C. König, A. Kraiss, and M. Sinnwell. Towards self-tuning memory management for data servers. *IEEE Data Eng. Bull.*, 22(2):3–11, 1999.
192. R. Wrembel. A survey of managing the evolution of data warehouses. *IJDWM*, 5(2):24–56, 2009.
193. X. Zhu, D. Young, B. J. Watson, Z. Wang, J. Rolia, S. Singhal, B. McKee, C. Hyser, D. Gmach, R. Gardner, T. Christian, and L. Cherkasova. 1000 islands: an integrated approach to resource management for virtualized data centers. *Cluster Computing*, 12(1):45–57, 2009.
194. M. Ziauddin, D. Das, H. Su, Y. Zhu, and K. Yagoub. Optimizer plan change management: improved stability and performance in oracle 11g. *PVLDB*, 1(2):1346–1355, 2008.

6.2 Summaries

D.J. Abadi, S.R. Madden, N. Hachem: Column-Stores vs. Row-Stores: How Different Are They Really? SIGMOD, 2008

(Summary by Robert Wrembel)

Two the most popular storage systems applied in practice for the ROLAP implementation of a data warehouse (DW) include a row-store and a column-store. Columnstore systems offer better performance of queries that access subsets of columns and compute aggregates. A column-store system can be simulated in a row-store system by: vertically partitioning tables, creating indexes - an index for one column used by a query, and creating materialized views that contain columns required by queries.

The paper compares the performance of a column-store and a row-store DWs, represented by C-Store and System X, respectively. In particular, the paper focuses on: (1) evaluating the performance of a row-store DWthat simulates a column-store DWby the three aforementioned mechanisms, and (2) demonstrating the impact of column-store query optimization techniques on a system's performance. The optimization techniques addressed in the paper include: late materialization (joining columns to form rows is delayed as much as possible in a query execution plan), block iteration (multiple values from a column are passed as a block to a query operator), compression techniques (e.g. run-length), and a new join algorithm (called invisible join) contributed in this paper.

The basic idea of the invisible join is to rewrite joins between a fact table and dimension tables into predicates on foreign key columns of a fact table. By using a has function for each predicate, bitmaps are constructed. Bits having values equal to 1 represent fact rows that fulfill a given predicate. By AND-ing the bitmaps, a query optimizer obtains the final bitmap describing fact rows fulfilling all predicates.

The experimental comparison of the aforementioned techniques was done on the star schema benchmark (a lightweight version of the TPC-H schema with a scale factor equal to 10). The obtained results for the row-store and column-store DW led the authors to the following observations. First, by applying vertical partitioning to a rowstore DW performance slightly improves but only when less than 1/4th of columns are selected by a query. It results from the implementation of vertical partitioning in row-store systems where row headers are included in every partition increasing its size. Moreover, combining column values into rows requires expensive hash joins. Second, while the indexing technique is used for all queried columns, the query optimizer of a row-based DW constructs an inefficient execution plan with hash joins. Third, the application of materialized views yields the best performance since appropriately constructed materialized view contains the required (possibly precomputed) data and joins are not necessary. Fourth, from the column-oriented optimization techniques the most significant ones include compression and late materialization. Finally, the experiments also showed that the denormalization of a star schema not always increases performance of a column-based system. Such a characteristic is caused by a query optimizer that applies the very efficient invisible join technique on a normalized star schema.

D.J. Abadi, S.R. Madden, N. Hachem: Column-Stores vs. Row-Stores: How Different Are They Really? SIGMOD, 2008

(Summary by Harald Schöning)

The paper covers two approaches. First, the authors try to show that column-store performance cannot be achieved by just configuring row stores in a column-store like manner. Experiments are limited to read-only OLAP scenarios. Three different setups of row stores are evaluated (using an anonymized commercial row store): 1) a maximum vertical partition of tables into two-column tables (table key, attribute) - no indexes defined as far as one can tell from the paper. Storage requirements for each of these two-column tables are significantly higher than for the corresponding columns in column stores, because of tuple header overhead, table key (where the mere position is sufficient in column stores) and missing compression (run-length encoding). The cost of tuple reconstruction is, however, introduced even for columns that are not restricted by predicates. Hence, it is not a surprise that performance of this variant is not convincing. 2) Answering queries index-only, even for those columns that are not restricted by predicates. It remains unclear why the authors did not use a combination of predicate-by-index evaluation and table lookup for query resconstruction. The introduced some optimizaion to the tuple reconstruction by extending the index key with the value of another column (e.g. the foreign key of a dimension table) but it seems there were still a lot of full index scans (which of course lead to bad performance) 3) creating a dedicated tailor-made materialized view for each query that contains only the columns needed for the query, apparently with some bitmap indexes defined. As a result, a full table scan is required. Interestingly, this method did not only exhibit the best performance. When the authors forced the column store to act similarly (by forcing the content of a row into a single column), performance of the column store was much worse than that of the row store In almost all cases, the materialized view variant showed the best performance, immediately followed by the "traditional" unmodified row store My personal conclusion on this part of the paper: I am not sure that the authors found the optimum configuration a row store allows for. As a second experiment (based on C-Store), various column-store specific optimizations are removed and one new is added to assess their influence on performance 1) Column-specific compression techniques such as run-length encoding are disabled. Obviously, organization of columns is position based (as opposed to value-ordered). Obviously, this causes more IO overhead and prevents the usage of operations on compressed data. A performance difference of factor 2 is reported 2) the late materialization is removed, i.e. tuples are reconstructed before any further processing. As a consequence, some tuples are reconstructed that are discarded by later processing steps, operations cannot work on compressed columns, cache performance decreases and block iteration cannot be applied (factor 3) 3) block iteration is removed - processing happens in a tuple-by-tuple mode. Array-processing, which benefits from modern CPU architectures, cannot be used then. With block iteration, C-Store performs 5%-50% better. 4) A novel join optimization is added, that mainly consists of rewriting a contiguous range of join values as range condition. As a side result, the paper argues that denormalization is not beneficial for column stores and decreases performance.

Self-tuning Histograms: Building Histograms without Looking at Data A. Aboulnaga and S. Chaudhuri Sigmod 1999.

(Summary by Jayant Haritsa)

This paper represents a pioneering effort in successfully applying autonomic computing concepts to database engines. Specifically, rather than the standard hard-wired approach of building a certain type of histogram (e.g. equi-width, equi-depth) for approximating attribute value distributions, an algorithm for allowing the histogram shape to incrementally adapt itself to the underyling data is proposed. Beginning with a uniform histogram, the learning process is implemented through a feedback process based on the estimation errors observed during query execution - the errors are assigned to the buckets in proportion to their current frequencies, and the bucket frequencies are either refined or their ranges reorganized based on these errors. The attractiveness of the self-tuning scheme lies in its not incurring the computational overheads associated with directly scanning the data, but gradually inferring the distribution through the metadata that is produced "for free" during query executions. This indirect approach also ensures that the histogram construction effort is independent of the size of the data, a particularly compelling advantage in today's world of tera- and peta-byte sized databases that need to be operational on a continuous basis. Finally, the self-tuning scheme naturally lends itself to efficiently building multi-dimensional histograms, a long-standing bugbear in the development of high-quality optimizer models.

M. Ahmad, A. Aboulnaga, S. Babu: Query Interactions in Database Workloads. DBTest, 2009

(Summary by Robert Wrembel)

The paper addresses the problem of measuring the performance of a database system for workloads composed of multiple concurrent queries. Since the queries interact with each other with respect to the subset of data accessed and the resources used, the performance measures may differ for different execution orders of the queries. The authors demonstrated this claim by experiments. To this end, they used the TPC-H benchmark on 1GB and 10GB databases, run by DB2 v.8.1. In the experiments the authors used various database workloads composed of TPC-H queries Q1 to Q22. A workload was composed of the sequence of the so-called query mixes. A query mix consists of a number of different number of instances of each query type, where different instances of a query type may have different parameter values.

The experiments focused on measuring: (1) overall system performance for different query mixes and (2) measuring resource consumption in different query mixes. The experiments showed that some interactions between queries in a mix decreased overall processing time while other interactions increased overall processing time. The experiments showed also that when queries run concurrently, resource utilization and competition can vary from one query mix to another. From the experiments, the authors drew two conclusions. First, that measuring a systemŠs performance without taking into account query interactions may lead to inaccurate conclusions on performance. Second, that there is no clear impact of a query mix on resource consumption.

In order to appropriately understand a system's performance obtained for a query mix, a DBA should know the workload in advance and the model of interactions between queries in the workload. Unfortunately, practically, it is impossible to achieve.

In order to be able to handle the impact of query mixes on a system's performance, the authors proposed to build the model of a workflow and query interactions. To this end, a statistical model is was proposed. This model is represented by the set of functions, $y = f(x1, x2, ..., xn)$ each of which yields the performance characteristic (e.g., CPU, I/O, overall query execution time) for a specific query mix, where $xi(i = 1, 2, ..., n)$ is the number of queries of type i in the mix. Function f() depends on a statistical model used. In the paper, the authors tested the accuracy of a linear regression and Gaussian processes. The experiments showed that the Gaussian processes provided a more accurate estimation of a performance characteristic.

Towards a Robust Query Optimizer: A Principled and Practical Approach Brian Babcock, Surajit Chaudhuri SIGMOD Conference 2005: 119-130

(Summary by Nico Bruno)

The main idea in the paper is to use sampling as the mechanism to obtain cardinality estimates. Using samples, then, we can obtain a probability distribution using Bayes' rule rather than a single expected value for a given predicate. Then, we can collapse such distribution in a different way depending on the degree of "robustness" that we require. We can see this "robustness" parameter as biasing selectivity values to the "right" (i.e., more tuples) when we require more robust answers, and to the "left" otherwise. In that way, we will consistently attempt to overestimate cardinality results in a principled way and avoid picking plans that can be very bad whenever the true selectivity is larger than its expected value from the underlying estimators.

Proactive Re-Optimization S. Babu, P. Bizarro and D. DeWitt Sigmod 2005.

(Summary by Jayant Haritsa)

It is well known that query optimizers often make poor plan choices because the compile-time estimates of various parameters, such as selectivities, are considerably at odds with those actually encountered at run-time. One approach to address this issue, which has been extensively investigated in the literature, is to dynamically reoptimize the query during the course of its processing, with the reoptimizations triggered by encountering significant errors in the parameter estimates. The runtime values replace these estimates in the reoptimization process, and may lead to a change in plans midway through the execution. Reoptimizations and plan changes can incur considerable computational overheads and loss of prior work, motivating this paper to propose an algorithm called Rio that attempts to minimize these overheads. Specifically, in the initial optimization phase, Rio uses a set of uncertainty modeling rules to classify selectivity errors into one of six categories, ranging from "no uncertainty" to "very high uncertainty", based on their derivation mechanisms. Then, these error categories are converted to hyper-rectangular error boxes drawn around the optimizerŠs point estimate. Finally, if the plans chosen by the optimizer at the corners of the principal diagonal of the box are the same as that chosen at the point estimate, then this plan is assumed to be robust throughout the box, and reoptimization is invoked only when the runtime errors are large enough to fall outside the box. Further, plans that are robust and easily switchable with a minimum loss of the work already done, are preferentially chosen during the optimization porcess.

"DIADS: Addressing the "My-Problem-or-Yours" Syndrome with Integrated SAN and Database Diagnosis" Babu, Borisov, Uttamchandani, Routray, Singh; FAST 2009

(Summary by Amol Deshpande)

Given a complex end-to-end system that includes a database system and a storage area network, the authors address the problem of identifying the root causes of query slowdown. Because of the complex interactions between the different components, it is not easy even for humans to quickly diagnose such issues. The authors attempt to address this problem in a systematic fashion by using monitoring data and event logs in conjunction with a symptoms database that encodes domain knowledge and machine learning techniques, to identify the possible reasons for the query slowdown.

The paper addresses a very important and tough problem and in my opinion, the key contributions of the paper lie in formalizing the problem and formulating an approach. By necessity, the approach requires a fair amount of up front work and is specific to a specific installation (this may one of the bigger challenges in applying this approach to different setups). The database-related causes are comparatively easier to identify, e.g., the plan may have changed causing an unintentional slowdown. However, the SANrelated causes are somewhat harder to pin down. For instance, we need to know which physical and logical components can have an impact on a specific operator in the query plan, and these components would be different for different operators.

Another component of the system is a symptoms database, which allows mapping the specific low-level inferences to high-level root causes. This database also needs to be populated externally using domain knowledge. The authors observe that several efforts are underway to create such databases in the industry. Finally the paper contains an extensive experimental study that uses a production SAN environment to illustrate the effectiveness of the system at identifying query slowdown causes. Overall, I think this is a great effort at solving an important problem, and should be studied further.

G. Candea, N. Polyzotis, and R. Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. PVLDB, 2(1):277-288, 2009

(Summarized by Jens Dittrich)

About: This paper presents a new approach to processing star queries in a data warehouse. Background: Traditional query processing engines use a query-at-a-time model to translate incoming declarative queries to a physical execution plan. However this approach neglects that concurrent queries may compete for resources invalidating the assumptions made by the cost-based optimizer in the first place. As an example consider multiple queries using the same I/O-device, e.g. a hard disk. Assume each query performs a large scan. Therefore, in terms of the cost model we would expect access times of two random I/Os plus the sequential scan costs. In reality, however, both queries will be competing for the hard disk. Each query will be allowed to execute one portion of the scan, then we switch to the other query triggering another random I/O and so forth. Overall this process could trigger considerable random I/O.

Solution: the authors propose a fixed physical execution pipeline. The pipeline uses a shared circular scan [Harizopoulos et al, SIGMO 2005]. It continuously scans the central fact table. Each tuple that is read by the circular scan is enriched by a bit vector. That bit vector contains one bit for each query in the workload to signal whether that tuple is relevant for a query. Each tuple is then joined (probed) to a set of hash tables where each one contains the union of the tuples of a dimension table satifying the search predicates of the current query workload. Again each tuple in each dimension is enriched by a bit vector signaling whether that tuple is relevant for a query. A probe of a tuple from the fact table then combines both bit vectors with a boolean AND. The effect is somewhat similar to a SIMD operation: data for multiple queries is computed with a single operation. Tuples with bit vectors where all bits are set to zero are dropped from the pipeline. After all tuples have been filtered, they are distributed to aggregation operators that group and aggregate the tuples.

Possible Impact: a very interesting solution to multi-query optimization. A nice and useful (actually I want to say "cool") extension of prior work, e.g. circular scans [Harizopoulos et al, SIGMO 2005] which was restricted to a single table; and "clock scans" which was built for a large denormalized table [Unterbrunner et al, PVLDB 2009]. In a way, this paper turns the star schema query processing problem into a streaming problem.

Identifying robust plans through plan diagram reduction Harish D., Pooja N. Darera, Jayant R. Haritsa PVLDB 1(1): 1124-1140 (2008)

(Summary by Nico Bruno)

The paper is the latest in a line of ideas around plan diagrams. A plan diagram for a parametric input query is a pictorial diagram that assigns a different color to each different plan that the optimizer picks for a given combination of input selectivity values. It was originally shown that plan diagrams are helpful in understanding the plan space of optimizers and also established that "anorexic" plan diagrams can be valuable. Anorexic diagrams are similar to plan diagrams, and assign a given plan to each discrete combination of input selectivity values. However, unlike traditional plan diagrams, in anorexic plan diagrams the total number of plans is minimized given the constraint that at each selectivity point, the cost of the chosen plan is at most Delta times the cost of the optimal plan at such point. Anorexic plan diagrams result in a small number of different plans for the whole spectrum of selectivity values without degrading the overall quality compared with the original plan diagram.

The paper extends the notion of anorexic plan diagrams taking into account cardinality estimation problems. The idea is that at runtime, the true selectivity value might be different from the estimated one. For that reason, the choice of plan (by using the plan diagram) would be done in the "wrong" area of the true selectivity space. This is a problem because (naturally) sometimes a plan p1 that are chosen using an anorexic plan diagram can be much worse than the original optimizer plan p2 in the region outside of that for p1 (due to cardinality errors).

The problem statement in the paper is to find a subset of the original plan diagram with minimum size, such as the plan assigned to each selectivity value is either the one picked by the optimizer or instead one that is no worse than Delta times compared to the original one picked by the optimizer *for every selectivity value*. The problem is NP-Hard and the paper explores heuristics to obtain such minimal plan diagrams and relies on some simplifying assumptions on the cost model of an optimizer. There is also an alternative problem formulation that makes additional assumptions on the cost model, which guarantee that only a small subset of points (those at "corners" in the space) need to be tested. This results in a faster technique with small drops in quality.

Adaptive Query Processing

Amol Deshpande, Zachary Ives and Vijayshankar Raman

This survey (at least the first part) is a must for every attendee of the seminar. The overall goal of this survey is to introduce into the concept to adaptive query processing, outlining the overall challenges, providing a general framework for classification of different approaches and finally discussing special aspects of adaptive query processing concerning multiple selection ordering and multiple join ordering. Finally, the survey closes with a list of open issues and potential research areas. The survey covers a lot of material which will be outlined as bullet points below; in order to be able to jump directly into the paper, we preserve the general structure of the paper.

Introduction

- fundamental breakthroughs of Codd's relational data model: convert a declarative, logic-based formulation of a query into an algebraic query evaluation tree
- Challenge: how to optimize regardless of where or how the data is laid out, how complex the query is, and how unpredictable the operating environment is.
- break-down of the System R-style optimize-then-execute model
  - optimizer error begins to build up at a rate exponential in the size of the query

- consequence of more general data management (querying autonomous remote data sources, supporting continuous queries over data streams, querying XML data, ...)

Adaptive query processing (AQP)

- addresses the problems of missing statistics, unexpected correlations, unpredictable costs, and dynamic data
- sing feedback to tune execution

Spectrum of AQP

- multiple query executions or adapt within the execution of a single query
- affect the query plan being executed or the scheduling of operations within the plan
- for improving performance of local DBMS queries, for processing distributed and streaming data, and for performing distributed query execution

Query Processing in Relational Database Systems

- conventional method: parse the SQL statement, produce a relational calculus-like logical representation, invoke the query optimizer
- query plan: a tree of unary and binary relational algebra operators (specific details about the algorithm to use and how to allocate resources; includes low-level "physical" operations like sorting, network shipping, etc.
- query processing model established with the System R project: divide query processing into three major stages.
  - Statistics generation is done offine on the tables in the database (information about cardinalities and numbers of unique attribute values, and histograms)
  - Query optimization: similar to traditional compilation; uses a combination of cost estimation and exhaustive enumeration
  - Query execution: virtual machine or interpreter for the compiled query plan
    - pipeline computation: full/ partial (multiple segments, materializing (storing) intermediate results at the end of each stage and using that as an input to the next stage)
    - scheduling computation
      - classical approach: iterator architecture: each operator has open, close, and getNextTuple methods
      - alternate approach: data-driven or datafow scheduling - data producers control the scheduling (aach operator takes data from an input queue, processes it, and sends it to an output queue)

Motivations for AQP: Many refinements to basic query proessing technology: mor epowerfull CPUs -> more comprehensive search of the space, relying less on pruning heuristics; selectivity estimation techniques have become more accurate and consider skewed distributions/attribute correlations

- Unreliable cardinality estimates: many real-world settings have either inaccurate or missing statistics or correlations between predicates can cause intermediate result cardinality estimates to be off by several orders of magnitude
- Queries with parameter markers: precomputed query plans for such queries can be substantially worse than optimal for some values of the parameters
- Dynamically changing data, runtime, and workload characteristics: queries might be long-running, and the data characteristics and hence the optimal query plans might change during the execution of the query
- Complex queries involving many tables: switch to a heuristic approach leads to estimation errors and the use of heuristics exacerbates the problem.
- Interactive querying: a user might want to cancel or refine a query after a few seconds of execution
- Need for aggressive sharing: traditional databases with almost no inter-query state sharing because their usage pattern is made up of a small number of queries against large databases.

Background: Conventional Optimization Techniques

Query Optimization: heart of cost-based optimization lies in selection ordering and join enumeration

Selection Ordering: refers to the problem of determining the order in which to apply a given set of commutative selection predicates to all the tuples of a relation

- Serial Plans: single order in which the predicates should be applied to the tuples of the relation
- Conditional Plans: generalize serial plans by allowing different predicate evaluation orders to be used for different tuples based on the values of certain attributes
- Static Planning: simple if the predicates are independent of one another; problem quickly becomes NP-Hard in presence of correlated predicates.
- Independent Predicates: optimal serial order can be found in O(nlog(n)) time by simply sorting the predicates in the increasing order of rank of the predicate
- Correlated Predicates: complexity depends on the way the correlations are represented (general: NP-Hard).
  $\Rightarrow$ The Greedy algorithm for correlated selection ordering
- Multi-way Join Queries: many choices: access methods, join order, join algorithms, and pipelining
  - Access Methods: pick an access method for each table in the query (direct table scan, a scan over an index, an index-based lookup on some predicate over that table, or an access from a materialized view.
  - Join Order: Def: is a tree, with the access methods as leaves; each internal node represents a join operation over its inputs. may not cover all join predicates, ie. if the join predicates form a cycle, the join order can only cover a spanning tree of this cycle

    - ⇒ picks a spanning tree over the join graph: eliminated join predicates are applied after performing the join, as "residual" predicates
  - Join Algorithms: nested loop join, merge join, hash join, etc.
  - Pipelining vs. Materialization: contain a blocking operator or not
    - + blocking operator: needs to hold intermediate state in order to compute its output, e.g., a sort / a hash join
      two classes:
      - + Non-pipelined plans: contain at least one blocking operator that segments execution into stages; Each materialization is performed at a materialization point
      - + Pipelined plans: execute all operators in the query plan in parallel
      - ⇒ offer very different adaptation opportunities
      - - for pipelined plans: changes in the tuple flow
      - - for non-pipelined plans: involve re-evaluating the rest of the plan at the materialization points

– Multi-way Join Queries: Static Planning
  - Projection, and in many cases selection, can be pushed down as a heuristic
  - fairly easy to develop cost modeling equations for each join implementation: given page size, CPU and disk characteristics, other machine-specific information, and the cardinality information about the input relations, the cost of executing a join can be easily computed
    - ⇒ challenge is to estimate the cardinality of each join's output result
  - in general, the cost of joining two subexpressions is independent of how those subexpressions were computed
    - ⇒ naturally leads to a dynamic programming formulation
  - impact of sorting
    - + a single sort operation can add significant one-time overhead
    - + may be amortized across multiple merge joins
    - ⇒ interesting order: sort order that might be exploited when joining or grouping

– Choosing an Effective Plan search limitation:
  - + deferred reasoning about Cartesian products until all possible joins were evaluated
  - + only considers left-deep or left-linear plans some extenions of modern optimizers:
  - + Plan enumeration with other operators: explore combinations of operators beyond simply joins. i.e. group-by pushdown
  - + Top-down plan enumeration: recursion with memoization with early pruning of subexpressions that will never be used: branch-and-bound pruning
  - + Cross-query-block optimization: allow for optimizations that move predicates across blocks and perform sophisticated rewritings, such as magic sets transformations

  + Broader search space: considers bushy plans as well as early Cartesian products. first partition the work using heuristics, and then run dynamic programming on each plan segment.
    selectivity estimation: given a set of input relations and an operation, the estimator predicts the cardinality of the result
  + System R: DBMS maintained cardinality information for each table, and selectivity estimation made use of this information, as well as minimum and maximum values of indexed attributes and several ad hoc "magic ratios" to predict cardinalities
  + Modern DBMSs: employ more sophisticated techniques, relying on histograms created offline to record the distributions of selection and join key attributes
- Robust Query Optimization: generell choice
  + between a "conservative" plan that is likely to perform reasonably well
  + a more aggressive plan that works better if the cost estimate is accurate, but much worse if the estimate is slightly off
    approaches: knob to tune the predictability of the desired plan vs. the performance by using such probability distributions Error-aware optimization (EAO): makes use of intervals over query cost estimates employ more sophisticated operators, for instance, n-way pipelined hash joins, such as MJoins or eddies. Such operators dramatically reduce the number of plans considered by the query optimizer, although potentially at the cost of some runtime performance.
- Parametric Query Optimization
  + An alternative to finding a single robust query plan: a small set of plans appropriate for different situations
  + postpone certain planning decisions to runtime
  + the simplest form uses a set of query execution plans annotated with a set of ranges of parameters of interest; just before query execution commences, the current parameter values are used to find the appropriate plan, e.g. choose-plan operators are used to make decisions about the plans to use based on the runtime information
  + not been much research in this area
  + commercial adoption has been nearly non-existent
  - big problem: what plans to keep around: the space of all optimal plans is super-exponential in the number of parameters considere relationship to progressive parametric query optimization the reoptimizer is called when error exceeds some bound or when there is no "near match" among the set of possible plan configurations
- Inter-Query Adaptivity: Self-tuning and Autonomic Optimizers
  + Several techniques: passively observe the query execution and incorporate the knowledge to better predict the selectivity estimates in future Examples: adaptive selectivity estimation: attribute distribution is approximated using a curve-fitting function Self-tuning histograms focus on general multi-dimensional distributions using histograms

Foundations of Adaptive Query Processing

- goal of adaptive query processing
  - + find an execution plan and a schedule that are well-suited to runtime conditions
  - + interleaving query execution with exploration or modification of the plan or scheduling space
  - ⇒ differences between various adaptive techniques can be explained as differences in the way they interleave Example: + System R-style: full exploration first, followed by execution
  - + Evolutionary techniques (e.g. choose nodes or mid-query reoptimization) interleave planning and execution a few times
  - + Radical techniques like eddies: not even clearly distinguishable.
- New Operators
  - + to allow for greater scheduling flexibility and more opportunities for adaptation
  - + requires greater memory consumption and more execution overhead, but the result is still often superior performance

1. Symmetric Hash Joins
   - + traditional hash join operator: must wait for the build relation to fully arrive before it can start processing the probe relation and producing results
   - ⇒ restricts adaptation opportunities since the build relations must be chosen in advance of query execution and adapting these decisions can be costly
   - + Symmetric hash join operator: build hash tables on both inputs ; when an input tuple is read, it is stored in the appropriate hash table and probed against the opposite table, resulting in incremental output
   - ⇒ enables additional adaptivity since it has frequent moments of points at which the join order can be changed without compromising correctness or without losing work
   - ⇒ disadvantage: memory footprint is much higher since a hash table must be built on the larger input relation
   - + Extensions: XJoin and the doubly pipelined hash join
   - ⇒ for a multi-threaded architecture, using producerŨconsumer threads instead of a dataflow model
   - ⇒ include strategies for handling overflow to disk
   - + Extensions: Ripple join: adapts the order in which tuples are read from the inputs so as to rapidly improve the precision of approximated aggregates
2. Eddy
   - + enable fine-grained run-time control over the query plans executed by the query engine
     - treat query execution as a process of routing tuples through operators

- adapt by changing the order in which tuples are routed through the operators (−> changing the query plan being used for the tuple)

+ eddy operator monitors the execution, and makes the routing decisions for the tuples.
  - routing table to record the valid routing destinations, and current probabilities for choosing each destination, for different tuple signatures.
+ Extension: SteMs; open the set of operators; not predefined at startup of query procesing
+ Pipelined operators offer the most freedom in adapting and typically also provide immediate feedback to the eddy
+ blocking operators are not very suitable

- Various auxiliary data structures are used to assist the eddy during the execution; broadly speaking, these serve one of two purposes:
  + Determining Validity of Routing Decisions: some form of tuple-level lineage, associated with each tuple, is used to determine the validity of routing decisions
    * use the set of base relations that a tuple contains and the operators it has already been routed through (→ tuple signature) as the lineage.
    * two bitsets : done and ready to encode the information about the operators that the tuple has already been through, and the operators that the tuple can be validly routed to next
- Implementation of the Routing Policy: refers to the set of rules used by the eddy to choose a routing destination for a tuple among the possible valid destinations
  + classified into two parts:
    - Statistics about the query execution: eddy monitors certain data and operator characteristics
    - Routing table: stores the valid routing destinations for all possible tuple signatures
- for probabilistic choices: a probability may be associated with each destination
- two-step process for routing a tuple:
  ++ Step 1: uses the statistics to construct or change the routing table (per-tuple basis, or less frequently)
  ++ Step 2: The eddy uses the routing table to find the valid destinations for the tuple, and chooses one of them and routes the tuple to it deterministic routing: O(1) probalisitic routing: O(H(p)) using a Hu?man Tree, where H(p) denotes the entropy of the probability distribution over the destinations

3. n-ary Symmetric Hash Joins/MJoins
   - generalizes the binary symmetric hash join operator to multiple inputs by treating the input relations symmetrically

- allows the tuples from the relations to arrive in an arbitrary interleaved fashion
- Idea: MJoins build a hash index on every join attribute of every relation in the query MJoin uses a lightweight tuple router to route the tuples from one hash table to another (use of eddy op) when a new tuple arrives:
  - ⇒ built into the hash tables on that relation
  - ⇒ porbe the hash tables corresponding to the remaining relations in some order to find the matches for the tuple (order in which the hash tables are probed is called the probing sequence)

- Adaptivity Loop
  - + regular query execution is supplemented with a control system for monitoring query processing and adapting it
  - + Adaptive control systems are typically componentized into a four-part loop that the controller repeatedly executes
    - ∗ Measure: An adaptive system periodically or even continuously monitors parameters that are appropriate for its goals, e.g. measuring cardinalities at key points in plan execution
      - · at the end of each pipeline stage
      - · add statistics collectors (i.e., histogram construction algorithms)
    - ∗ Analyze: Given the measurements, an adaptive system evaluates how well it is meeting its goals, and what is going wrong
      - · determining how well execution is proceeding - relative to original estimates or to the estimated or measured costs of alternative strategies
      - · the only way to know precise costs of alternative strategies is through competitive execution, which is generally expensive
      - · all adaptive strategies analyze (some portion of) past performance and use that to predict future performance
    - ∗ Plan: Based on the analysis, an adaptive system makes certain decisions about how the system behavior should be changed
      - · often closely interlinked with analysis
      - · changing the query plan requires additional "repairs"
      - + Query scrambling may change the order of execution of a query plan
      - + Query plan synthesis may be required
      - + Corrective query processing requires a computation to join among intermediate results that were created in different plans: "cleanup" or "stitch-up" phase
    - ∗ Actuate: After the decision is made, the adaptive system executes the decision, by possibly doing extra work to manipulate the system state.
      - · cost depends on how flexible plan execution needs to be
      - · some previous work may be sacrificed, accumulated execution state in the operators may not be reused easily and may need to be recomputed

        · simplest case: query plans can only be changed after a pipeline finishes: actuation is essentially free, but prior work might have to be discarded
        · plan change in the middle of pipelines execution: make sure that the internal state is consistent with the new query plan
          – different approaches: only consider switching to a new query plan if it is consistent with the previously computed internal state
          – to switch at certain consistent points indicated by punctuation markers
          – to use specialized query operators to minimize such restrictions, e.g., MJoins, SteMs
          – to modify the internal state to make it consistent with the new query plan (dynamic plan migration)
          – Example: STAIR operator: exposes the internal state using APIs and allows to change the internal state, for ensuring correctness or for improving the performance

– Post-mortem Analysis of Adaptive Techniques
  - adaptive techniques may use different plans for different input tuples $\rightarrow$ makes it hard to analyze or reason about the behavior of these systems
  - goal of "post-mortem" analysis: understand how the result tuples were generated and, if possible, to express the query execution in terms of traditional query plans or relational algebra expressions.
  - two types of behavior:
    + Single Traditional Plan: use a single traditional plan for all tuples of the relations, with the only difference being that the plan is not chosen in advance
    + Horizontal Partitioning: splitting the input relations into disjoint partitions, and executing different traditional plans for different partitions

– Adaptivity Loop and Post-mortem in Some Example Systems
  - System R
    + Measurement: measures the cardinalities of the relations and some other simple statistics on a periodic basis
    + Analysis & Planning: statistics are analyzed during planning at "compile" time
    + Actuation: straightforward, when the query is to be executed, operators are instantiated according to the plan chosen
    + Post-mortem: System R uses a single plan, chosen at the beginning of execution, for all tuples of the input relations
  - Ingres (one of the earliest relational database systems)
    + highly adaptive query processing, no notion of a query execution plan; instead it chose how to process tuples on a tuple-by-tuple basis
    + Example: join the data from n tables, R1,...,Rn, the query processor begins by evaluating the predicates on the relations and materializing

the results into hashed temps by a special one variable query processor (OVQP) query processor begins by choosing a tuple from the smallest table values from this tuple are substituted into the query resulting (new) query is recursively evaluated process continues until a query over a single relation is obtained, which is then evaluated using OVQP the decision at each step of recursion is made based on the sizes of the materialized tables → different plans may be used for different tuples

  + Measurement: sizes of the materialized tables that get created during the execution.
  + Analysis & Planning: done at the beginning of each recursive call (after a call to OVQP)
  + Actuation: by substituting the values of a tuple as constants in the current query to construct a new (smaller) query
    –> Ingres query processor interleaves the four components of the adaptivity loop to a great extent
  + post-mortem analysis: tricky, because Ingres does not use traditional query operators

- Eddies: unify the four components of the adaptivity loop into a single unit and allow arbitrarily interleaving
  + Measurement: can monitor the operator and data characteristics at a very fine granularity
  + Analysis & Planning: are done with the frequency decided by the routing policy, e.g. lottery scheduling for every tuple
  + Actuation: process and cost depend largely on the operators being used
  + Post-mortem: behavior can usually be captured using traditional query plans and horizontal partitioning, although this depends on the rest of the operators used during execution.

Adaptive Selection Ordering

– Adaptive Greedy : continuously monitors the selectivities of the query predicates using a random sample over the recent past, and ensures that the order used by the query processor is the same as the one that would have been chosen by the Greedy algorithm
  + Analysis: continuously by the reoptimizer, which looks for violations of the greedy invariant using the matrix-view
  + Planning: If a violation is detected in the analysis phase, the Greedy algorithm is used to construct a new execution plan.
  + Actuation: The stateless nature of selection operators makes plan switch itself trivial
  + Post-mortem: The query execution using the A-Greedy technique can be expressed as a horizontal partitioning of the input relation by order of arrival, with each partition being executed using a serial order. The

A-Greedy technique is unique in that it explicitly takes predicate correlations into account, and analytical bounds are known for its performance. No other technique that we discuss has these properties

- Adaptation using Eddies
  + Routing Policies
    - Deterministic Routing with Batching
    - Lottery scheduling
    - Content-based Routing

Adaptive Join Processing: Overview

... based on the space of the execution plans they explore

- History-Independent Pipelined Execution
- History-Dependent Pipelined
- Non-pipelined Execution

Adaptive Join Processing: History-Independent Pipelined Execution

- Pipelined Plans with a Single Driver Relation
  - Static Planning
    * Choosing Driver Relation and Access Methods
    * Join Ordering: Reduction to Selection Ordering
  - Adapting the Query Execution
- Pipelined Plans with Multiple Drivers
  - n-ary Symmetric Hash Joins/MJoins
  - Driver Choice Adaptation
  - State Modules (SteMs)
    * Query Execution and Hybridization
    * Routing Constraints
  - Post-mortem Analysis
- Adaptive Caching (A-Caching)

Adaptive Join Processing: History-Dependent Pipelined Execution

- Corrective Query Processing
  - Separating Scheduling from Cost
  - Post-mortem Analysis
  - A Generalization: Complementary Joins
  - Open Problems
- Eddies with Binary Join Operators
  - Routing Policies
  - Post-mortem Analysis
  - Burden of Routing History
- Eddies with STAIRs
  - STAIR Operator
  - State Management Primitives
  - Lifting the Burden of History using STAIRs
  - State Migration Policies
  - Post-mortem Analysis
- Dynamic Plan Migration in CAPE

Adaptive Join Processing: Non-pipelined Execution

- Plan Staging
- Mid-Query Reoptimization
  - Flavors of Checkpoints
  - Switching to a New Plan
  - Threshold to Invoke Reoptimization
  - Computation of Validity Ranges
  - Bounding Boxes and Switchable Plans
- Query Scrambling
- Summary and Post-mortem Analysis

Summary and Open Questions

- Trade-Offs and Constraints
  - Precomputation vs. Robustness vs. Runtime Feedback
  - Data Access Restrictions
  - Exploration vs. Exploitation
  - Optimizer Search Space
  - Plan Flexibility vs. Runtime Overhead
  - State Management and Reuse
- Adaptive Mechanisms
- Challenge Problems
  - Understanding the Adaptive Plan Space
  - Developing Optimal Policies
  - Effective Handling of Correlation
  - Managing Resource Sharing
  - Scaling to Out-of-core Execution
  - Developing Adaptation Metrics

K. E. Gebaly, A. Aboulnaga: Robustness in automatic physical database design. EDBT, 2008

(Summarized by Robert Wrembel)

The paper presents the approach to developing a database advisor for index design. The work is motivated by the fact that existing advisors (both commercial and prototype ones) produce advices that may cause deterioration of overall system's performance. First, because estimated benefits obtained from additional physical data structures, suggested by an advisor, may be inaccurate. This inaccuracy often results from wrong cardinality estimation due to the lack of multi-column statistics on correlated columns. Second, because a training workload used for advising index designs is often slightly different than future real workloads, executed after creating the recommended indexes.

For these reasons, the authors proposed an approach to develop a database advisor, called Multi-Objective Design Advisor that will be more reluctant to the aforementioned problems. To this end, the authors proposed to define two new metrics, namely Risk and Generality, for the evaluation of the quality of a proposed index design. The Risk metric measures how resistant is the index design to errors made by cost estimation module of a query optimizer. In other words, the metric quantifies risk of selecting an index that will not improve (or deteriorate) a query performance. Risk is expressed by means of the current cost of executing a workload and an estimated maximum cost (the worst case). The Generality metric measures how well an index design supports changes to a workload. The metric is expressed by means of the number of unique index prefixes being recommended (the higher the number of unique prefixes the better the Generality).

The incorporation of these new metrics into a database advisor requires changes in the optimization objective function. The authors defined this function as a weighted sum of Benefit, Risk, and Generalization, where Benefit represents a performance benefit of a traditional advisor.

In order to prove the applicability of this approach, series of experiments were conducted on a TPC-H benchmark database (scale factor 1) run on PostgreSQL. The experiments showed that: (1) an advisor supporting the Risk metric performs better than a traditional advisor and (2) the Generality metric allows to recommend the set of indexes that support also workflows that differ from training workflows.

Database Cracking

Stratos Idreos, Martin L. Kersten, Stefan Manegold CIDR 2007: 68-78 *(Summarized by Nico Bruno)*

The paper presents an intriguing idea towards self-managing database organization. Rather than manually building indexes in a single shot, the idea is to piggyback on top of traditional query processing and progressively build indexes on columns that are interesting. At a very high level, the idea is similar to that of quicksort in main memory, where we iteratively choose a pivot value and shuffle around elements that are smaller or larger than the pivot.

Consider a system that stores data column-wise, where each column consists of pairs (value, rid). Suppose there is a query that asks for all tuples satisfying a¡10. the first time we scan the corresponding column and filter the values that satisfy the predicate. In addition to this step, we additionally use 10 as a "pivot" value and create 3 partitions while scanning: one that contains tuples with a¡10 (in random order), another with a=10, and another with a¿10. We then store these partitions using a tree-based structure for fast processing of partition information. Suppose that we have a new query with predicate a¿20. To evaluate such query, we can then safely ignore the partition that contains tuples with a¡10 and a=10, and only scan the one with a¿10 (in doing so we create 3 partitions, one with 10¡a¡20, a=20, and a¿20). As we keep working in this way, we progressively build a full index that only requires accessing the right data (or very close to the minimum amount of data). Initially searches are slowsince we scan large portions of data, but (i) they are still faster than waiting for a index to be built, and (ii) the "indexes" are quickly built, on demand, over columns that are relevant during query processing. The paper introduces the idea of cracking, discusses several implementation aspects, how the ideas relate to indexing and sorting, how to extend the approach for row-oriented databases, several experimental results on a prototype, and many interesting follow-up ideas (e.g., handling updates or join predicates). These last two ideas are further explored in subsequent papers.

Database Cracking

Stratos Idreos and Martin L. Kersten and Stefan Manegold *(Summarized by Stratos Idreos)*

Database systems can provide optimal performance only for the workload they were tuned for. Through the use of auxiliary data structures and proper data layout they can prepare for the anticipated queries. This works perfectly if two conditions are met: (a) the workload is known upfront and (b) there is enough idle time to prepare the system with the proper structures and physical design before the first query arrives. A plethora of seminal work advanced this area of research providing solutions on how we can design automatic tuning advisors that can off-line analyze an expected workload and propose the proper physical design.

But what if the workload is not known upfront? Then, we can rely on solutions that provide on-line tuning, i.e., during the first few queries, an online tool monitors the system performance and eventually it provides a recommendation for a proper physical design.

But what if the workload is not stable? What if the workload changes again before we even had the chance to analyze it or before we even had the chance to exploit the indices we built?

This paper proposes a research path towards such dynamic and unpredictable environments where there is little if any time to invest in physical design preparations while at the same the workload cannot be safely predicted.

The idea is based on online and continuous physical reorganization. Data is continuously reorganized by query operators using the query requests as an advise of how data should be stored. This paper provides an initial architecture in the context of columnstores, showing new selection operators and plans that reorganize columns based on the selection predicates. The net effect can be seen as an incremental quick sort, i.e., every query will perform only 1 or 2 steps of a quick sort on a column, leaving behind a column that has more structure which future queries can exploit. Areas of the column that are not interesting for the workload are never optimized while performance quickly reaches optimal levels for the hot parts of the column.

Updating a cracked database

Stratos Idreos and Martin L. Kersten and Stefan Manegold *(Summarized by Stratos Idreos)*

(Requires reading summary for paper "Database Cracking. CIDR07") With continuous data reorganization during query processing and within query operators, comes the challenge of updates. Updates imply physical changes on a column which might conflict with the physical changes of database cracking. The goal is to maintain the performance benefits even under updates.

Various approaches are studied. The simplest approach is the one of dropping any auxiliary data structure upon a conflict. The next query that requires this column, needs to make a new copy of the column and start reorganizing it from scratch. This means that periodically we loose all knowledge gained by previous queries.

The best approach found in this paper, is that of self-organizing differential updates. Incoming updates are kept as pending updates until a relevant query arrives, i.e., a query that requires the given value of the pending update. The algorithm then, on-the-fly merges the pending value within the column performing the less possible physical actions in its effort to maintain a physically contiguous column. The updates logic is attached within the selections operators and performed during query processing. It exploits the property that a cracking column contains multiple pieces in sorted order but within each piece there is no order. The best algorithm will even move values from the normal column to the pending updates to make room for the needed values of this query, i.e., make sure the column is consistent for the current query only and worry for the rest in the future and only on demand.

S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. CIDR, pages 68-78, 2007

(Summarized by Jens Dittrich)

About: follow-up paper to [Kersten and Manegold, CIDR 2005]. Contains more technical detail on how to implement cracking in a column store (MonetDB).

Solution: the main idea is to copy (replicate) each original column the first time it is being touched by a where-clause of a query. Then that column (a so-called cracker column) is gradually reorganized as described in [Idreos, Kersten, and Manegold, CIDR 2007]. The original columns are preserved. Columns not touched by where-clauses are not replicated. Like this the cracking (pay-as-you-go indexing might be a more appropriate name for this) builds a set of adaptive indexes for each table. Internally, each cracker column implements cracking by reordering data inside the column, i.e. there is no breaking of columns into pieces but simply a partial re-sort of data inside the column. Overall, columns referenced heavily will more and more approach the structure of a fine-granular index, whereas columns referenced rarely will only create a coarse-granular index (which corresponds to a horizontal partitioning).

Possible Impact: an interesting solution to adaptive indexing in a column store. A nice step towards self-managing database systems. Prof. Jens Dittrich Chair of Information Systems Group Saarland University Campus, E1 1, 220.1, Tel. +49 (0)681 302-70140 http://infosys.cs.uni-saarland.de

Self-organizing tuple reconstruction in column-stores

Stratos Idreos and Martin L. Kersten and Stefan Manegold *(Summary by Stratos Idreos)*

(Requires reading summary for paper "Database Cracking. CIDR07") This paper studies the problem of multi-attribute queries in database cracking. It demonstrates the problem of performance degradation due to random access patterns in tuple reconstruction. For example, when selecting a few tuples from column A and then we need to get the corresponding tuple from column B, e.g., to perform an aggregation, then we are essentially performing a join operation on the positions (rowids). Column-store architectures rely heavily on sequential access patterns and positional joins to provide efficient tuple reconstruction. Database cracking though, reorders data causing random data access and thus bad performance, as the paper shows, during tuple reconstruction.

The solution comes via self-organizing tuple reconstruction operators that work also via data reorganization as opposed to joins. In other words the columns, we need to reconstruct are adaptively reorganized in the same way as the columns we select on, ensuring sequential access patterns over aligned column areas.

The paper also demonstrates solutions for dealing with storage restrictions. A single column consists from multiple smaller columns instead of a contiguous area. Each smaller column is independently created, reorganized, aligned, and dropped if storage is limited. As before everything happens adaptively and on-demand within cracking operators during query processing, requiring no human administrators, idle time or preparations.

M. Ivanova, M. L. Kersten, and N. Nes. Self-organizing strategies for a column-store database. In EDBT, pages 157168, 2008.

(Summary by Harald Schoening)

The idea of this paper is to partition columns in a column store based on the real query workload.

The paper addresses read-only workloads only and assumes value-based column organisation as provided by MONETDB's binary association table (BAT).

Even when the columns are value-based, a full column scan is necessary to evaluate range queries. Segmenting the column reduces IO cost when the query can be restricted to a few or even one segment. The idea is to use the actual query workload as splitting criterion, in particular range queries. As too many splits would lead to too small segments, two splitting policies are considered, one being random-based and one based on ranges for segment sizes Splitting can be done either 1) as post processing which does not put the reorganization burden on the query execution, but on the other hand misses the chance to reuse query results, or 2) by eager materialization, which implies that parts of the column that are not needed for query execution nevertheless have to be materialized because of splitting or 3) by lazy materialization (also called adaptive replication). In this case, the range addressed by the query is materialized in addition to the larger original segment(hence leading to replicaed data). The paper mentions that control of the number of replicas is essential but does not provide a convincing solution.

The paper claims that the presented form of self-reorganization is beneficial compared to static segmentation and/or replication but the experiments do not address this claim. The experiments also do not address the querstion whether there is any benefit for using the ranges occurring in real queries as splitting boundaries, considering that in the end all segements will be of more or less the same size.

M. L. Kersten and S. Manegold. Cracking the database store. CIDR, pages 213224, 2005.

(Summarized by Harald Schoening)

This paper addresses the horizonal splitting (cracking) of tables based on query workload. In this respect it is similar to the paper "Self-organizing Strategies for a Column-store Database". However, in this paper the usefulness of the idea is not only discussed for a column store but also for row stores. The idea is to use predicates in queries (in particular range expressions and equi-join predicates) as "cracking" criteria. Since for a row store, queries can address disjoint attribute sets in their predicates, split occur in varying dimensions. The resulting cracking index (that keeps track of the segemtns that have been created) resembles some multi-dimensional access structures as proposed in the early 90s, but the paper does not elaboraze on this. Many questions on cracking index policies (size and number of segments etc.) are mentioned as open problems. The application area forseen for database cracking are OLAP and scintific databases. Here, some multi-query patterns are addressed by experiments: homerun (drill-down), hiking and strolling. The experiments are done fo rMonetDB (column store) and two open-source row stores, but with 2-column tables only (which does not lead to a dimension mix). The conclusion is that database cracking is not beneficial for current row-store offerings.

M. L. Kersten and S. Manegold. Cracking the database store. In CIDR, pages 213-224, 2005

(Summarized by Jens Dittrich)

About: proposes a new approach to adaptive indexing. Background: the appropriate choice of indexes speeds-up may speed-up query processing considerably. The right choice of indexes, however, have to be made anually by a DBA or with the help of an automatic index selection tool. Solution: this paper takes a different approach. The idea is to interpret each incoming query as a hint how to break the database into pieces. For instance a simple select * from R where R.aą10 may be interpreted to break the database into two sets of tuples: one containing all tuples where R.aą10 and a second set for R.a£=10. This reorganization of the database store is termed "cracking"'. Each query (may) trigger(s) a reorganization requests with extra costs, i.e. the costs for the reorganization are piggybacked onto the query. If many queries come in, this process corresponds to a lazy index creation: the more a table was "cracked" the more fine-granular the index becomes. That index gradually adapts to the workload. It also considers skew, i.e. values refrenced more frequently will trigger a more fine-granular index [also compare Dittrich et al, SIGMOD 2005, who consider such an adaptive (de-)index organization in the presence of updates]. The paper discusses several issues when implementing this idea in an existing relational DBMS with a focus on a columnar storage manager. Possible Impact: an interesting solution to adaptive indexing. See also follow-up work [Idreos, Kersten, and Manegold, CIDR 2007]

"Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors"; Moerkotte, Neumann, Steidl; VLDB 2009

(Summary by Amol Deshpande)

The area of selectivity estimation has seen much work, both practical and theoretical, over the last 20 years, but the more important question of how much it matters has seen much less work. Quantifying the effect of one wrong selectivity estimate on the overall plan chosen is nearly impossible to do, since that depends on the rest of the query, and the errors in the rest of the estimates. In some cases, the errors may not matter because they don't change the relative order of the plans, in other cases, the errors may cancel out.

This paper studies this issue and presents the first theoretical results on the problem. They define the notion of q-error, a form of multiplicative error metric, to capture the selectivity estimation errors. Consider a single histogram that tries to approximate a set of points (x i, b i), where x i denote the different values of attribute, and b i denotes the frequency. Say the estimated value for x i is denoted by f approx(x i). Then q-error is maximum over all i, the value of: max(b i/f approx(x i), f approx(x i)/b i). The authors prove several interesting results using this. For instance, for cost functions that follow the ASI property (which allows us to use the "rank ordering algorithm"), the authors can show some tight results. They also obtain some results for more general cost functions. Although the results are impressive and first of a kind, the practical utility of the theorems is unclear. One of the results shows a q4 factor difference between optimal and produced plans. However, q is defined to be the maximum multiplicative errors over all intermediate and base relations, which itself may hide an exponential term on the estimation errors on base relations.

Perhaps the key utility of the analysis is to motivate q-error, and ask the question whether we can build synopsis structures that minimize q-error directly. The second part of the paper focuses on this issue, and develops a deep algorithm for minimizing q-error for certain classes of histograms. I firmly agree with the authors these types of histograms should be used in place of histograms that minimize average error (not the least because we also tried to minimize the multiplicative error in our work on Dependency-based Histograms in SIGMOD 2001 :).

Extreme Visualisation of Query Optimizer Search Spaces A. Nica, D. Brotherston and D. Hillis Sigmod 2009

(Summary by Jayant Haritsa)

This paper describes a functional tool, SearchSpaceAnalyzer (SSA), for visualizing the plan search space of an industrial-strength database query optimizer (SQL Anywhere from Sybase). In contrast to most other optimizers, which typically employ a breadth-first approach to identify the best plan, SQL Anywhere chooses to use a depthfirst enumeration that prunes partial plans costlier than the best complete plan identified thus far. This technique is reminiscient of the historical "pilot-plan" strategy proposed by Rosenthal, Dayal and Reiner about two decades ago - while attractive in principle, the scheme had proved to be impractical since the pilot plan, which was essentially randomly chosen, failed to provide a meaningful degree of pruning. This issue of pilot plan choice is addressed in SQL Anywhere through a proprietary algorithm whose objective is to enumerate access plans in a carefully chosen order that maximizes the likelihood of producing cheap plans early in the enumeration exercise.

The SSA visualizer supports the entire spectrum of views of the search space, ranging from high-level summaries to detailed diagrams of subspaces, representing the plans in a highly compressed manner without sacrificing the essential optimization characteristics. It can be used to analyze the differences between search spaces generated by the optimizer on different platforms, as well as under different system loading conditions, or with different optimization levels.

SSA is complementary to other visualizer tools such as Picasso in that it characterizes the input search space where Picasso characterizes the output parametric optimality space.

“Predictable performance for unpredictable workloads,” P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann; PVLDB, 2(1):706-717, 2009

(Summary by Amol Deshpande)

The goal of this paper is to design a system that has very good yet predictable performance (e.g., with latency and freshness guarantees) under a diverse and unpredictable workload, containing both read queries and updates. The key proposed technology is a distributed relational table implementation, called Crescando. Crescando combines the idea of "collaborative scans", recently studied in several works, in a very interesting manner with "query indexes" (from data streams literature). Being scan-based, Crescando does not use indexes on the data itself, but rather the incoming queries are batched and indexed (to the extent possible). The relations (or relation partitions) are continuously scanned, and matched against this index to generate result tuples. At any point, thousands of queries may share a scan cursor on a relation. For selective queries, this can result in huge savings over an approach that matches every query against every tuple. Crescando uses a new scan algorithm, called Clock Scan, that simultaneously has a write cursor and a read cursor on the relation segment.

One key feature of Crescando is that, it uses essentially the same ideas for supporting a large number of "updates" against the data. Guaranteeing ACID properties is not easy in this case, but the authors are able to develop an approach to handling that. The authors also discuss how to partition the relations, which types of indexes to build, how to deal with non-uniform memory architectures and so on.

The authors have implemented most parts of the Crescando system, and comparisons with alternative approaches indicate that the proposed approach can provide guaranteed query and update throughput for a wide range of workloads. As the authors also acknowledge, Crescando mainly focuses on handling simple (single-table) queries and updates, and is not necessarily appropriate in all scenarios. However, I believe that there are many use cases for such a system. It is also an interesting future direction to generalize the types of queries handled while providing the same types of guarantees.

R. Wrembel: A survey of managing the evolution of data warehouses. IJDWM, 5(2):24-56, 2009

(Summarized by Robert Wrembel)

Existing methods of designing a data warehouse (DW) usually assume that a DW has a static schema and structures of dimensions. Unfortunately, this assumption in practice is often false. Managing the evolution of a DW is challenging from a research and technological point of view.

This paper surveys problems and solutions to managing changes/evolution of a DW. The issues are illustrated with the MultiVersion Data Warehouse approach, developed by the author. In this approach, a DW evolution is managed by the MultiVersion Data Warehouse (MVDW) that is composed of a sequence of its versions, each of which corresponds either to the real-world state or to a what-if scenario. The described MVDW approach provides the following: (1) a query language capable of querying possibly heterogeneous DW states and capable of querying metadata on an evolving DW, (2) a framework for detecting changes in EDSs that have an impact on the structure of a DW, (3) a data structure based on bitmaps for sharing fact or dimension data by multiple DW versions, (4) index structures, i.e., ROWID-based multiversion join index and bitmap-based multiversion join index, for the optimization of the so-called star queries that address multiple DW versions.

7 Participants

Name	**Affiliation**
Agrawal, Parag	Stanford University
Ailamaki, Anastasia	Ecole Polytechnique Fédérale de Lausanne (EPFL)
Al-Omari, Awny	HP Neoview
Bruno, Nicolas	Microsoft Research
Chaudhuri, Surajit	Microsoft Research
Cole, Richard	ParAccel
Deshpande, Amol	University of Maryland
Dittrich, Jens	Saarland University
Ewen, Stephan	Technische Universität Berlin
Graefe, Goetz	HP Labs
Haritsa, Jayant	Indian Institute of Science
Idreos, Stratos	Centrum Wiskunde & Informatica (CWI)
Ilyas, Ihab Francis	University of Waterloo
Kemper, Alfons	Technische Universität München
Kersten, Martin	Centrum Wiskunde & Informatica (CWI)
König, Arnd Christian	Microsoft Research
Krompass, Stefan	Technische Universität München
Kuno, Harumi	HP Labs
Lehner, Wolfgang	Technische Universität Dresden
Lohman, Guy	IBM Almaden Research Center
Manegold, Stefan	Centrum Wiskunde & Informatica (CWI)
Markl, Volker	Technische Universität Berlin
Mitschang, Bernhard	Uni Stuttgart, Germany
Neumann, Thomas	Technische Universität München
Nica, Anisoara	Sybase
Paulley, Glenn	Sybase
Poess, Meikel	Oracle
Polyzotis, Neoklis	UC Santa Cruz
Salem, Ken	University of Waterloo
Sattler, Kai-Uwe	Technische Universität Ilmenau
Schöning, Harald	Software AG
Simon, Eric	SAP
Waas, Florian	EMC Greenplum
Wrembel, Robert	Poznań University of Technology