Automated Query Interface for Hybrid Relational Architectures

DISSERTATION

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von Dipl. Inform Andreas Lübcke

geb. am 06.02.1982 in Havelberg

Gutachterinnen/Gutachter

Prof. Dr. Gunter Saake
Prof. Dr. Thomas Neumann
Prof. Ladjel Bellatreche (PhD)

Magdeburg, den 24.02.2017

Lübcke, Andreas:
Automated Query Interface for Hybrid Relational Architectures
Dissertation, Otto-von-Guericke-Universität Magdeburg, 2017.

Zusammenfassung

Grundlegende Ziele der Datenhaltung sind das effiziente Speichern von sowie der effiziente Zugriff auf Daten. Datenbanksysteme (DBSs) müssen dabei unterschiedliche Anforderungen für verschiedenste Anwendungen erfüllen. Dafür wird die Datenhaltung für die Anwendungen mit Hilfe von diversen Optimierungstechniken für DBSs maßgeschneidert. Eine mögliche Optimierung für DBSs ist der physische Entwurf mit Hilfe von speziellen Datenstrukturen (z.B. Indexe). Diese Datenstrukturen verbessern hauptsächlich den Zugriff auf die Daten. Einerseits sollen DBSs verschiedene Anwendungen für das Tagesgeschäft sowie für Vorhersagen unterstützen. Andererseits haben verschiedene Anwendungszwecke unterschiedliche Anforderungen und Entwurfsziele, wodurch ein optimaler Entwurf für mehrere Anwendungszwecke innerhalb eines DBS nicht möglich ist. Darüber hinaus wird die Komplexität des Entwurfs durch die Abwägung verschiedener Entwurfsziele deutlich erhöht. Als Konsequenz widersprechen sich Mehrzweck-DBS und optimaler Entwurf mit praktikabler Komplexität für einen bestimmten Anwendungszweck. Verschiedene Anwendungsgruppen (d.h., Domänen) werden von Forschern unterschiedlichen Entwurfsregeln zugeordnet, denn unterschiedliche Domänen zeigen verschiedene Zugriffsmuster auf Daten. Für die unterschiedlichen Zugriffsmuster existieren unterschiedliche Speicherarchitekturen, welche jeweils für verschiedene Domänen besser geeignet sind. Dem folgend, sind zeilenorientierte Datenbanksysteme (Row Stores) am besten für schreiblastige Domänen geeignet, das heißt für das Tagesgeschäft, wohingegen spaltenorientierte Datenbanksysteme (Column Stores) für leselastige Domänen wie Vorhersagen (d.h., Analysen) geeignet sind. Die Speicherarchitektur kann im Gegensatz zu Datenstrukturen nicht einfach angepasst werden, wenn sich die Anforderungen an das DBS ändern. Weiterhin gibt es für die Auswahl einer Architektur wenig Werkzeugunterstützung, so dass Datenbankdesigner sich anhand von Heuristiken vorab für eine Architektur entscheiden. Neben der Schwierigkeit zukünftige Anforderungen vorhersagen zu können, beschreiben Forscher, dass die verschiedenen Domänen nicht disjunkt zu einander sind. Das führt für Datenbankdesigner zu dem Dilemma, dass keine Architektur alle Domänen optimal unterstützt.

Unser erstes Ziel ist daher die Werkzeugunterstützung für die Architekturauswahl für einen Anwendungszweck zu schaffen. Das Dilemma der Architekturauswahl für sich verändernde Anforderung abzuschwächen ist unser zweites Ziel. Dies bezieht sich auch auf Anwendungszwecke, welche sich zwischen den klassischen schreib- und leselastigen Domänen einordnen. Für die werkzeuggestützte Architekturauswahl erweitern wir bestehende Ansätze für den physischen Entwurf wie sie für Row Stores bekannt sind. Diese Ansätze ermöglichen eine Analyse der Domänenanforderungen und schlagen Datenstrukturen für die entsprechende Domäne vor. Einen derartigen

Ansatz integrieren wir in unseren Advisor für den physischen Entwurf, welcher uns eine Auswahlentscheidung anhand von Kostenschätzungen (d.h., Datenbankstatistiken) und von Stichproben ermöglicht. Dabei schließen wir die semantische Lücke bei der architekturspezifischen Bewertung von Anfragen (z.B. bei der Kostenschätzung) durch einen neuen Extraktions-, Normalisierungs- und Abbildungsansatz. Bei der Entwicklung des Advisors für den physischen Entwurf haben wir herausgefunden, dass die verschiedenen Architekturen sich auch in ihren angestammten Domänen nicht im Allgemeinen gegenseitig übertreffen (z.B. bei Anfragezeiten). Um auf diesen Sachverhalt zu reagieren und sich ändernde Anforderungen besser unterstützen zu können, integrieren wir unseren Ansatz in einen hybriden Speicher, der beide Speicherarchitekturen unterstützt (d.h., Row Store und Column Store). Darüber hinaus führen wir eine Schnittstelle für hybride Architekturen ein, die zwei unterschiedliche Varianten unterstützt. In der ersten Variante sind zwei Systeme enthalten (d.h., ein Row Store und ein Column Store); wohingegen die zweite Variante durch unseren Prototypen repräsentiert wird, der beide Architekturen in einem System unterstützt. In verschiedenen Fallstudien haben wir beobachtet, dass keine Architektur die andere für leselastige, schreiblastige oder dazwischenliegende Domänen gänzlich übertrifft. In der ersten hybriden Speichervariante entspricht die Verteilung von Aufgaben im abstrakten Sinne einem Aufbau mit jeweils einem System pro Domäne. Lediglich der Zugriff auf die aktuellsten Daten ist in dieser Variant möglich. Nichtsdestotrotz haben wir einen Hybridspeicher entwickelt um die Vorteile einer kleinteiligeren Verteilung von Aufgaben zu zeigen. Dabei haben wir Column Store Funktionalität in einen bestehenden Row Store integriert. Mit Hilfe unseres Prototyps konnten wir zeigen, dass Hybridspeicher (a) für gemischte (Lese-/Schreib-) Domänen geeignet sind, und (b) dabei konkurrenzfähige und im gewissen Maße außerordentliche Ergebnisse für diese Domain liefern. Als Konsequenz unserer Ergebnisse sollten neue Architekturen, die zwischen den beiden klassischen Architekturen positioniert sind, zukünftig für den physischen Entwurf und die Anfrageverarbeitung in Betracht gezogen werden.

Abstract

Basic goals of data management are the efficient storage and the efficient access to data. Thereby, Database Systems (DBSs) have to meet different requirements for different applications. With several optimization techniques for DBSs, data management is tailored for different applications. One option for optimization is the physical design of DBSs using support structures, which mainly improve access to data (e.g., indexes). On the one hand, DBSs shall support multiple purposes for daily operation as well as forecasts (*one system fits all*); on the other hand, different purposes imply differing requirements and design goals, which prevent an optimal design for each purpose, and increase the complexity of physical design. In other words, both goals – multi-purpose DBSs and optimal design (for one purpose) with feasible complexity – contradict each other (*one system does not fit all*). Researchers describe that different groups of applications (i.e., domains) match different sets of design rules due to different access pattern to data. For different access pattern, different database (storage) architectures exist, which are most suitable for conflicting domains. Row-oriented Database Management Systems (Row Stores) are most suitable for a write-mostly domain – daily operations; whereas Column-oriented Database Management Systems (Column Stores) are most suitable for a read-mostly domain – forecasts. In contrast to support structures, storage architectures cannot easily adapt to changing requirements, and architecture decision lacks of tool support. Thus, database designers decide with heuristics for certain architecture beforehand. Beside complexity of requirement prediction, researchers describe that both domains are not disjoint. Consequently, database designers face the dilemma that architectures do not optimally support either domains or intermediate domains.

Our first goal is to cope with the dilemma of design decision with tool support for both domains. Second, we want to mitigate the design dilemma for changing requirements and for intermediate domains (e.g., frequent change of storage architecture). For the design dilemma, we extend existing approaches for physical design of Row Stores, which already allow to analyze domain requirements and to advise sets of support structures for certain domain. We integrate these approaches in a physical design advisor for storage architectures, which allows decision based on cost estimates (i.e., DBS statistics) as well as on samples. We cope with the gap of architecture-specific requirements concerning cost estimates with a new cost extraction, normalization, and representation approach. Hence, our design advisor utilizes this approach to compute architectural decision. Beside the tool support for design decisions, we showed that architectures do not outperform each other for their inherent domains in general. For changing requirements or intermediate domains, we integrate our approach to hybrid stores, which support both architectures

– Row Stores and Column Stores. We introduce the Automated Query Interface for Hybrid Relational Architectures and discuss two different hybrid setups, which the query interface supports. The first setup is composed of two separate systems – a Row Store and a Column Store; whereas the second setup is our prototype, which supports both architectures (i.e., in one system). We observe in case studies that neither storage architecture outperforms the other for read-mostly, write-mostly, nor intermediate domains. Besides access to most up-to-date data, a distribution of tasks in hybrid stores with two systems resembles setups with disjoint DBSs for each domain. To show benefits for more fine-grained distribution of tasks, we implement a hybrid store, in which we add Column Store functionality to an existing Row Store. With our prototype, we showed that hybrid stores (a) are suitable for intermediate domains and (b) show at least competitive, in some extent outstanding results for such domains. As a result, new architectures, which are in between steady architectures, have to be considered for physical design as well as query processing.

Acknowledgments

Doing research and writing dissertation is a lot hard work. In the beginning, there is so much research to recognize and then there is too less time to consider all the new ideas. Hence, it is hard to be focused on research goals and then there are these unexpected issues that want to be solved. It does not get easier during research progress; believe me. Finally, it is very hard to find the (best) point to finish the thesis – the hardest part in my opinion. To come to the point, my gratitude counts to all individuals who have accompanied me during recent years, thank you all! Nevertheless, I want to thank some of you individually.

First, I thank my wife, my family, and my son for their eternal support and patience, who gave me strength and tenacity over and over find the track to finish this thesis. And not forgetting, my friends who helped me to take my mind off the work and to overcome the one or two low point.

Second, I thank Gunter Saake who makes possible my research at the University of Magdeburg in the first place. Early on, Gunter integrated me as tutor and student researcher in his working group, and later on gave me free choice for my research topic I am interested most. Gunter always had confidence in my work, nevertheless, as my supervisor he always had an advice in tough situations. I enjoyed the time in your group and being part of it; thank you!

Third, I thank my advisor Veit Köppen as well as my coauthors for their assistance, for issuing challenges to me, and helpful discussions on my research. I particularly thank to Ingolf Geist, Martin Kuhlemann, and Martin Schäler. Moreover, I express my gratitude to the members (i.e., former and current) of the Database and Software Engineering Group for the pleasant and fruitful atmosphere – particularly to my roommates Ingolf, Eike, and Reimar; it was an enjoyable job.

Fourth, I would like to thank Thomas Neumann, who gave me helpful advice at early stages of my research. Thank you for the fruitful discussions.

In the end I would like to thank my students.

Contents

List of Figures

List of Tables

List of Abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
AMPL	A Mathematical Programming Language
AST	Automatic Summary Table
AQUA²	Automated Query Interface for Relational Architectures
CPU	Central Processing Unit
Column Store	Column-oriented Database Management System
C/R	Compression Ratio
DBMS	Database Management System
DBS	Database System
DDL	Data Definition Language
DM	Decision Model
DML	Data Manipulation Language
DSM	Decomposition Storage Model
DWH	Data Warehouse
ETL	Extract Transform Load
HDBMS	Hybrid Database Management System
HDBS	Hybrid Database System
HDD	Hard Disk Drive
I/O	Input and Output
MAUT	Multi-Attribute Utility Theory
MCDA	Multi-Criteria Decision Analysis
MCDP	Multi-Criteria Decision Problem
NSM	n-ary Storage Model

OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
OS	Operating System
QGM	Query Graph Model
Row Store	Row-oriented Database Management System
SEQUEL	A Structured English Query Language
SQL	Structured Query Language

1. Introduction

Data management has a long history as software starts to process data and is ever since a complex task. Early data management approaches were highly dependent on internal representations and implementation details steered by the purpose of software [LM67]. Hence, Codd [Cod70] proposes a higher level data representation based on relations, which aims on a general-purpose data management. First Database Management Systems (DBMSs) were implemented based on the ideas of Codd – Row-oriented Database Management Systems (Row Stores) [SHWK76, ABC$^+$76]; whereby tuples represent entities of data objects. Therefore, the idea was: store entities of objects together, thus, tuples (or rows of a table) are physically stored as unit.

Researchers determined very early that Database Systems (DBSs) need (index) structures to improve performance and support a variety of applications (e.g., inverted files [DL65], combinatorial hashing [Riv76], or B^+-tree [Knu73, Pages 559 ff.] or [Com79]). The sum of (index) structures for a DBS is named as physical design. The variety of index structures and applications led to the dilemma of optimal physical-design selection which is well-known as index-selection problem nowadays [Sch75, HC76, Com78, SAC$^+$79, ISR83, CFM95, GHRU97, CDN04]. This problem passes to the architectural design of DBS through the development of storage strategies which decompose tuples into their single domain (i.e. column of a table) in extreme case and are also known as Column-oriented Database Management System (Column Store) nowadays [WFW75, THC79, CK85]. These approaches cannot prevail against existing DBMSs at this time, thus, Row Stores persist as general-purpose DBMSs. This fact led to specialization of DBSs for a certain purpose (e.g., a system for either daily operations or analyses) along with development of design advisor for different purpose DBSs [VZZ$^+$00, KLS$^+$03, ZZL$^+$04, ZRL$^+$04, ACN00, ACK$^+$04, BC06, BC07].

Row Stores dominate the field of data management until mid-2000s when researcher started questioning the general-purpose approach (i.e., one size fits all) [Sc05]. At the same time, Column Stores came to life again [MF04, SAB$^+$05, ZBNH05, AMF06, AMDM07, Aba08, SBKZ08, ABH09]. Column Stores turn out being suitable for analyses (i.e., read-only/-most workloads), in which they outperform Row Stores by orders of magnitude [SAB$^+$05, AMH08]. In other words, the previous specialization of DBSs correlates to the storage architecture now. However, physical design advisors persist as DBMS-specific or at least architecture-specific, thus, architecture selection is based on heuristics. In short, Row Stores are most suitable for daily operations; whereas Column Stores are most suitable for analyses. In consequence, nowadays the physical design starts with optimal storage-architecture

selection followed by physical design on a certain DBS/architecture for the same purpose (cf. index-selection problem). Whereas optimal selection of index structures is supported by tools, heuristic-based architecture selection reaches limitations by new requirements for mixed workloads (e.g., analyses on daily-operations data in real-time) [Van01, Lan04, ZAL08, SB08, SBKZ08].

In this thesis, we discuss approaches for physical design in relational data management which in their domain are applicable and sophisticatedly. Although the physical design for a certain DBMS is such complex that (close to) optimal configurations are only feasible with tool support. New opportunities for physical design due to different storage architectures raise the question if design space is inconveniently pruned by a-priori DBMS selection. We make this question free from confusion. Specifically, we transfer ideas from physical design tools and therewith give assistance for DBMS or storage-architecture selection. We question if DBMS or storage architecture is suitable for a given application and then we advocate physical-design tuning by existing tools (cf. also DBMS recommender by Brahimi et al. [BBO16]). As a follow-up challenge, we have to cope with higher complexity for physical design on the one hand; and on the other hand, we cope with adaptation for changing conditions, which takes advantage of the increased search space for physical design. Even for small workloads, we observe that storage-architecture selection becomes towards infeasible without closer insights and tool support as we present in this thesis.

1.1. Contribution

In this thesis, we analyze the impact of storage architectures for a given workload (domain) and propose an approach to overcome the architecture selection problem. First, we analyze basic approaches for relational data management and discuss their capabilities. In the following, we analyze widest spread storage architectures (i.e., Row Store and Column Store) with respect to their capabilities for a certain workload domain. While problems and benefits of architectures are discussed in detail in literature, benefits of hybrid architecture become obvious by comparison of both architectures for a mix of classic disjoint workload domains. Rather, we analyze the impact of storage architecture based on a wide-spread analytical benchmark.

Second, we propose, implement, and evaluate four key aspects that in combination result in our approach, which we name Automated Query Interface for Relational Architectures (AQUA$^{\mathbf{2}}$). In detail, we contribute the following four aspects:

1. Architecture-independent workload decomposition, representation, and normalization are necessary to compare storage architecture in-plane. Our approach – workload pattern – is straightforwardly derived from internal query representation in relation DBMSs and stores statistics from query optimizers respectively user samples with respect to architecture specifics. Statistic normalization is necessary due to internal different representation of statistics. Even though our approach is straightforward, it is necessary for the remaining aspects of our work.

2. Storage-architecture selection for given workloads based on our workload representation. We outline architecture selection on arbitrary degree of detail (i.e., in dependency of detail degree of statistics) that is transparent according to cost functions and thus, to cost criteria. Despite the statistic-based architecture selection, we propose selection approaches for unknown workload (parts) considering uncertainty and multi-criteria decision with user weights. Moreover, we derive heuristics for physical design from our experiments. Architecture selection is only applicable for DBS redesign or a-priori design on known workloads and user samples, but this work is crucial for remaining aspects of our work (e.g., query processing in hybrid storage systems). We depict the overall decision procedure in Figure 1.1.

3. A hybrid query interface (AQUA$^{\mathbf{2}}$) integrates our workload-representation and architecture-selection approaches. Within our query interface we analyze queries (i.e., parts of a workload) to decide where we execute these best on hybrid stores (e.g., redundant with both architectures). Therefore, we propose a stepwise optimization – rule-based on the global level and cost-based on the local (storage) level. For the rule-based optimization, we propose query-execution heuristics which we implement in our prototypical hybrid store.

4. Hybrid DBMS implementation with two architectures and query engines. We additionally implement Column Store functionality in an open-source Row Store. The integrated AQUA$^{\mathbf{2}}$ framework dispatches queries in the hybrid DBMS prototype rule-based. Hence, we evaluate our prototype with AQUA$^{\mathbf{2}}$ integration based on a mixed domain benchmark and show significant improvements in comparison to the original implementation. Finally, we outline potential improvements to our prototype and ideas for (a more-general) hybrid DBMS implementation.

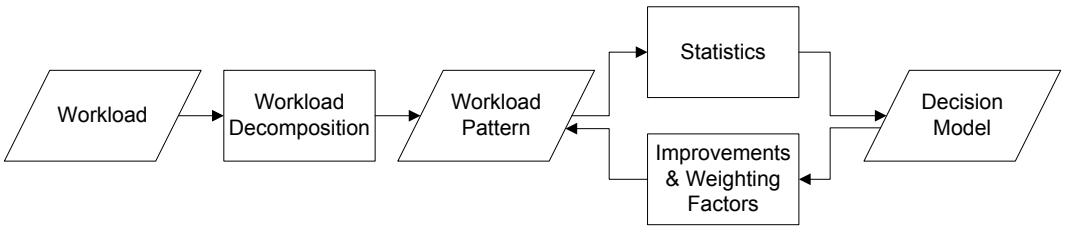


Figure 1.1.: *Workflow of the storage-architecture-decision process.*

1.2. Outline

In Chapter 2, we introduce general concepts of the relational data model, relational storage architectures, different application fields for relational DBMSs, and (potential) optimization methods. Hereby, we show the basis for our work and assist the

understanding and the classification of our work for readers who are unfamiliar with relational data-management concepts.

In Chapter 3, we survey two storage architectures for the relational data model and discuss their benefits and drawbacks. The survey shows necessity for storage-architecture selection as well as new challenges for either approach on mixed requirements for former disjoint application domains. In this context, we show a brief study for a certain application domain on both approaches that substantiates (design) challenges by ambiguous results of the study.

In Chapters 4 and 5, we propose, discuss, and evaluate our storage-architecture-selection approach. Therefore, we introduce an architecture-independent (workload) decomposition and representation approach in Chapter 4 which is able to store data of different systems simultaneously. In Chapter 5, we discuss cost estimation based on our workload representation that we combine and evaluate in a storage advisor for three different decision situations.

In Chapter 6, we present concepts for hybrid relational storage architectures. We introduce our query-interface framework, discuss different optimization methods, and present heuristics for hybrid systems. Subsequently, we show results of a mixed (requirement) workload on two different hybrid system setups – two replicated DBMSs and a prototypical hybrid DBMS.

In Chapter 7, we discuss related research compared to key aspects of our approach. Therefore, we compose our consideration to the following five key aspects: workload representation, self-tuning, physical design, relational storage architectures, and hybrid DBMSs.

In Chapter 8, we summarize our contributions and suggest directions for future research.

2. Background

In this thesis, we discuss architectural approaches for relational data management and the selection of those. For readers who are unfamiliar with relational data-management concepts, we give therefore a brief introduction into main (historic) concepts, application fields, optimization techniques, and their relevance in this chapter.

2.1. Relational Database Management Systems

In this section, we present fundamental ideas which led to the development of relational Database Management Systems (DBMSs) and prepare the ground of their success. In the following, we describe the relational data model [Cod70] itself.

Ever since software exists data has to be stored; therefore, engineers developed data-storage approaches (e.g., graphs represent objects and relationships also known as network model). Such data systems were based on low-level programming and had no common (higher level) interface. Moreover, they were often not compatible to each other even though they were deduced from the same abstract (data-organization) approach (e.g., tree-structured files). That is, query and administration of data were highly dependent on implementation details and internal representation (e.g., cf. Levien and Maron [LM67]).

In the 1970s, drawbacks by this type of data management were identified (e.g., by Edgar F. Codd [Cod70]); thus, data management came into research focus and first DBMSs were developed (e.g., Ingres[1] [SHWK76], System R [ABC+76, CAB+81]). Codd introduced a data-representation approach – the relational data model – based on relations [Cod70] that each represents up to n sets (cf. [Chi68a, Chi68b, Chi68c] for another set-based approach).

Relations are usually shown in an array representation (i.e., tabular; cf. Figure 2.1) even though the row order is irrelevant. A row represents an n-tuple of a relation R; whereby the column ordering is significant as it corresponds to the domain order of R (i.e., relations are domain-ordered). To hide order dependency from users, users can use domain-unordered relations. Therefore, columns (i.e., domains of R) have to be uniquely identifiable (e.g., by name). We identify n-tuple of R by one domain or by a combination of domains that is non-redundant and we call it primary key (e.g., `partkey`; cf. Figure 2.1). A primary key (or its parts) of a relation S can be a foreign key of R (e.g., `part_brand`; cf. Figure 2.2), thus, the foreign key of R describes the

[1] Announcement 1974: `http://engineering2.berkeley.edu/labnotes/1003/history.html`.

R	(partkey	partname	part_brand)
	1	conrod	Mahle
	2	wishbone	Meyle
	3	piston	Woessner

Figure 2.1.: *Array representation of a ternary relation.*

R	(partkey	partname	part_brand)	S	(part_brand	city)
	1	conrod	Mahle		Bosch	Stuttgart
	2	wishbone	Meyle		Mahle	Stuttgart
	3	piston	Woessner		Meyle	Hamburg
					Woessner	Weil am Rhein

Figure 2.2.: *Foreign key – A dependency between relation R and S.*

reference or dependency to the domain in S[2]. A domain (or their combination) is denoted as foreign key of R if it is not the primary key of R, but its elements refer to (parts of) the primary key of another relation S (e.g., `part_brand` is not the primary key in R but in S; cf. Figure 2.2).

A data model itself – even if it is implementation-independent – is not enough to compute stored data. An (abstract) implementation-independent processing scheme is needed as well. That is, low-level implementation is irrelevant as long as abstract result fits to the processing scheme (e.g., to obtain a domain directly via identifier versus read entire tuples and omit not needed domains). Therefore, Codd [Cod70] described the following operations on relations:

Permutation: Generates one permutation of R; whereas $n!$ permutations exist with subject to n domains of R. That is, domains of R are interchanged (e.g., converse of R). Permutation is usually transparent to the user due to the fact that each permutation can be generated by corresponding projection of all domains implicitly (see below).

Projection (π): Selects a certain subset of domains of a relation and omits the rest. The result itself is a relation (i.e., the projection of the given relation).

Restriction (Selection σ): Generates a subset of a relation (i.e., a new relation R') based on another relation S in the way that R' contains all n-tuple of R which satisfy equality to m-tuples of S with subject to: domains of $S \subseteq$ domains of R. In other words, S represents the result relation for a selection predicate, whereas predicate selection is directly computed over R nowadays (i.e., still results in R').

[2] Whenever we omit normal forms, foreign key may occur within relations [Cod70].

Cartesian product ($\times$): Is inherently given because each relation R is a subset of the Cartesian product of its domains.

(Natural) Join ($*$): Combines two relations R and S to a new relation J (cf. Figure 2.3) that have domains (at least one; e.g., k) in common. That is, J encloses all domains of R and S without duplication of mutual domains (i.e., k) and returns all combined tuples that satisfy equality of k in both relations (i.e., $\pi_k(R) = \pi_k(S)$). In general, natural join can be rewritten by the Cartesian product and subsequent selection and/or projection to represent a broad variety of joins.

Composition ($\circ$): Describes the combination of R and S without k (i.e., R following S); whereby R and S have to be joinable. Furthermore, projection eliminates duplicates from the join result (i.e., $R \circ S = \pi_{'m+n-k'}(R * S)$. However, the composition of R and S can be dependent on the performed join. We argue, this operation has no significant relevance in current DBMSs.

In [Cod72], Codd showed the (relational) completeness of the relational algebra – a collection of operations on relations. Codd included operations (e.g., union, difference) to his considerations that were inherently supported by definition of the relational data model, due to the fact that the model was mathematically defined on relations. Furthermore, Codd showed the completeness of the relational calculus – a more declarative way to query data. In contrast, queries are specified in a more procedural way in the relational algebra[3]. However, Codd showed that that the relational algebra and the relational calculus are equivalent in their expressive power (to query data) including set operations [Cod72].

The proliferation of Database Systems (DBSs)[4] – especially relational DBS – and an ever-increasing number of users asked for tools and query languages that were not only accessible for professionals. Therefore, Chamberlin and Boyce introduced A Structured English Query Language (SEQUEL) [CB74]. SEQUEL has a long history of improvements, adaptations, and derivations, which led de facto to standardization in form of Structured Query Language (SQL) in 1992 [Ame92, Int92, MS92]. With further development of SEQUEL, the borders softened between historical distinct database languages. That is, SQL is not only a query language (e.g.,

[3]Codd regarded relational calculus and algebra due to historical development of different query-language domains.

[4]A DBS paraphrases the combination of a DBMS and a database (i.e., the organized data itself).

$R * S$	(partkey	partname	part_brand	city)
	1	conrod	Mahle	Stuttgart
	2	wishbone	Meyle	Hamburg
	3	piston	Woessner	Weil am Rhein

Figure 2.3.: *Natural join of relations R and S.*

cf. [CAE+76, DD97, GP99, EM99]). SQL comprises additionally a Data Manipulation Language (DML), which changes data sets (i.e., insert, delete, or update data), and a Data Definition Language (DDL), which creates and/or modifies data schema (i.e., an instance of the data model). Other research showed also the high demand for plain implementation-independent data representations (e.g., the Entity-Relationship Model [Che76] that can represent a number of data models).

We assume for this thesis that the union-compatibility is constituted by set operations as well as renaming of attributes is constituted by the Cartesian product operator [KBL05, Page 133 ff.]. Further, we are able to derive other join types (e.g., with condition, outer joins) from a set of five basic relational operators [KBL05, Pages 137 ff.]:

1. the Projection π,
2. the Selection σ,
3. the Union $\cup$,
4. the Set difference $-$, and
5. the Cartesian product $\times$.

Finally, we highlight that the relational algebra and SQL are not equivalent and have different constraints. Relational DBMSs are not based on sets as the algebra is defined on, and thus, these systems need special integrity constraints (e.g., `UNIQUE`) and operations for duplicate elimination (e.g., `DISTINCT`). Further differences are:

- Row and column order are significant for SQL (e.g., sorted result respectively grouping is dependent on column order),
- Duplicate columns names (i.e., domain names of R) may occur as well as anonymous columns (e.g., unnamed columns after aggregation) in SQL syntax,
- Duplicate rows can occur, whereas duplicate tuples cannot.

A comprehensive overview to properties of relational-algebra operations can be found in [KBL05, Pages 127 ff.]. The corresponding considerations for SQL queries can be found in [KBL05, Pages 147 ff.]. Nevertheless, researchers recognized early that translation of SQL to algebra is helpful for proof of query semantic and equivalence as well as for query optimization (e.g., Ceri and Gottlob [RKB87]).

2.2. Relational Database Architectures

One of the major benefits of the relational data model is the independence from implementations details, thus, a number of different implementations is possible (i.e., architectures). A major aspect of our work is the distinction of relational architectures and their selection with respect to different parameters. In the following, we present therefore different implementations of the relational data model from history which evolved over the years.

The relational data model itself and relational schemas could be implementation-independent; nevertheless the data representation has to be transferred to physical storage layers for persistent storage (e.g., store data on Hard Disk Drive (HDD)). Usually, DBMSs do not have direct access to HDDs but mainly use interfaces from Operating System (OS) or storage-management tools. A common container (i.e., (minimal) DBMS-storage unit) for communication between DBMS and physical storage is the so-called *page*[5] which is often sized as a HDD-data block or a multiple of these (e.g., cf. [KBL05, Pages 322 ff.]). We highlight, storage organization within pages is only logical, whereas DBMSs have any freedom to organize their data within pages, which are transformed via interface for physical storage layer automatically.

We state, there are two main approaches for storage organization within pages. First, first DBMSs – known as Row-oriented Database Management Systems (Row Stores) – took over ideas from Codd [Cod70] to physical storage layout directly – the n-ary Storage Model (NSM) (cf. Section 2.1). That is, an n-tuple of a relation (i.e., a row) is a unit which is stored together. In other words, rows are stored one after the other (cf. Figure 2.4).

Second, transposed storage models [WFW75, THC79] were in research focus from the beginning of relational DBMS that later are summarized under the term Decomposition Storage Model (DSM) (e.g., a (full) DSM with surrogate keys [CK85]). These approaches store all values of the same domain (i.e., attribute) of a relation together (cf. Figure 2.4).

We argue, many approaches proposed mixed NSM/DSM layout that cluster most important data in one file (i.e., clustered transposed files; e.g., Batory [Bat79], March et al. [MS77, MS84]). Today, such approaches are known as (vertical) fragmentation (on a more abstract level) in distributed DBMS (e.g., cf. [EN10, Pages 894 ff.]. We note, these approaches do not necessarily correlate to the relational model because most approaches are based on (data) files (e.g., by Wiederhold et al. [WFW75]). However, several researchers compared the performance of DSM to NSM approaches (e.g., March et al. [MS77, MS84], Batory [Bat79], Cornell and Yu [CY90]). In these decades, the DSM approaches had their right to exist for special use cases (e.g., clinical and statistical/governmental data [WFW75, THC79]), but were not able to overcome their drawbacks for wider spread. Some of the drawbacks were redundant storage (e.g., [WFW75]) respectively increased total storage size [CK85] on limited Input and Output (I/O) resources, query processing were more complex (e.g., [Bat79]), physical design (i.e., the decomposition schema) were very complex and bothered with changing retrieval requirements [MS77, MS84, NCWD84], because decomposition mostly correlated to distribution on different HDDs for best performance with DSM approaches, and reduced update performance for DSM [CK85] which in comparison was highly dependent on the number of inverted files – state-of-the-art at this time – for comparable NSM solutions.

Nevertheless, until mid-2000s nearly all relational DBMS used a NSM approach

[5] In former systems, files are commonly the (minimal logical) storage unit.

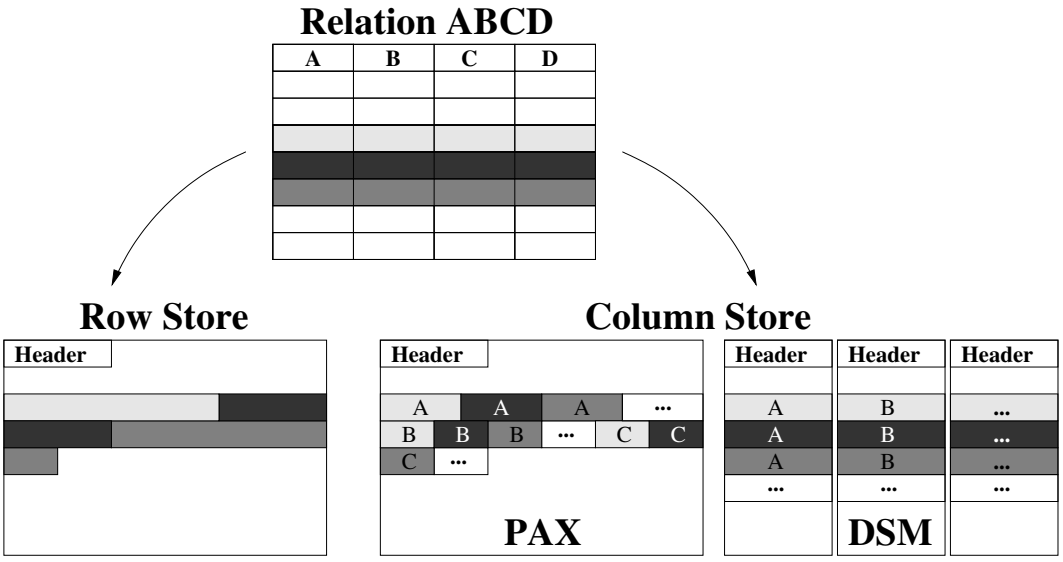


Figure 2.4.: *Exemplary storage organization of Row Stores and Column Stores within pages respectively files.*

(e.g., System R [ABC+76], INGRES [SHWK76], Oracle[6]) except for Sybase IQ (formerly Expressway), which implemented a DSM approach since the early 1990s. Modern DSM systems – known as Column-oriented Database Management Systems (Column Stores) (e.g., Sybase IQ [MF04]) – have no redundant storage and decompose the data fully (i.e., data is only stored domain-wise) what is highly different to former DSM systems. Some systems use writable storages or nodes to overcome update problems (e.g., Sybase IQ [How10, Syb10], C-Store [SAB+05, Aba08]) and from time to time compute something like a mini Extract Transform Load (ETL)-process (cf. Section 2.3).

As Copeland and Khoshafian already stated in [CK85], hardware and database technology had to highly evolve to support DSM approaches, which happened in the last decades (e.g., higher I/O bandwidth, cheaper main memory, improved HDD setups like RAID [PGK88]). Central Processing Units (CPUs) evolved as well, thus, DSM approaches could take advantage of new technologies (e.g., for joins [MBNK04], optimize cache consumption [ZNB08]). Moreover, (de-) compression is relatively less expensive with respect to CPU costs than the decades before, what advocates data compression in DBMS. The fully Decomposition Storage Model supports compression superiorly due to the fact that attributes have plain data type, thus, high Compression Ratios (C/Rs) can be achieved (e.g., Infobright ICE [Inf08, Inf11a]), whereas rows consist of several data types. That is, one of the major drawbacks – the increased total storage size – is reduced by superior compression support.

We argue, approaches occurred repeatedly which simulated either architecture

[6]Oracle – introduction and history: `https://docs.oracle.com/cd/E11882_01/server.112/e40540/intro.htm#CNCPT88783`.

in the other and compare them to each other (e.g., [CY90, AMH08]). In contrast, Ailamaki et al. presented a mixed NSM/DSM storage layout [ADHS01]. That is, data within a page (or a file) is domain-wise stored – as for the DSM; but all domains of a tuple are kept on the same page – as for the NSM. We show the coarse storage layout for PAX in Figure 2.4. In [Böß09], Bößwetter presents PAX for super pages (SPAX) that combines several pages to super pages. Data within super pages is domain-wise stored; whereby, access to super pages is organized with B^+ tree. Furthermore, data is separated by fix-length attributes and variable-length attributes within super pages. We state, these approaches have no direct impact on our work, because they aim at cache efficiency and do not affect I/O behavior. Furthermore, modern DSM systems have superiorly been established compared to PAX-like approaches, thus, we do not consider these in more detail here.

For DSM systems, the term Column-oriented Database Management Systems (Column Stores) has been established. Nevertheless, PAX-like approaches have as well a column-oriented storage layout, thus, we do not specifically distinguish between these. That is, we use the term Column Store only. DBMSs, which use the NSM, are named as Row-oriented Database Management Systems (Row Stores) in the following.

2.3. Application Fields

We discussed the relational data model and different architectures, which indicated to different application fields for relational data management. Hence, we discuss the categorization of applications and their characteristics in the following. The distinction of application fields implies the necessity to analyze applications for best possible support by data management, which is part of the motivation of our work. In the beginning, we introduce terms, which are necessary for the discussion later on. Subsequently, we discuss the origin of different application fields and their relation to each other.

In previous consideration, we discussed several relational operators (cf. Section 2.1), which enclosed tasks (i.e., database operations) for the DBS implementation-independent. A single relational operator does not provide a satisfying result to the user because operators produce or compute intermediate results only. Hence, a sequence of operators is needed which automatically computes (intermediate) results. Therefore, we summarize a sequence of relational operators with the term *query* whenever results are returned from database. DBSs respond to various queries of different type, thus, a single query does neither qualify the application sufficiently nor the requirements to the DBS. Moreover, queries can be arbitrary complex (e.g., single tuple lookup – a point query – versus queries with multiple joins). Therefore, we characterize a set of queries as *workload*. Various sets of queries can differently characterize workload, thus, workloads are summarized to workload types. These workload types reflect the core characteristics of workloads (i.e., the application field) with respect to their requirements to the DBS.

In history (i.e., 1960s/70s), a database workload was known beforehand due to antecedent usage of the batch-processing model. For this processing model, it was necessary to know all tasks which were pooled in a batch; and therefore, one optimized database (and their physical design) for certain (type of) batches. We argue, a workload classification was not necessary for batches (from present point of view) because one was able to react on new batches beforehand.

In the early 1980s, the situation dramatically changed by wider spread of ATMs[7]. To support ATMs, daily end-user transactions needed to be processed and thus, (financial) transaction processing had to be automated in the corresponding DBSs (i.e., online processing). This trend advanced two developments. First, workloads were not known in advance anymore and second, new ideas evolved for correctness of automated concurrent computation – especially Atomicity, Consistency, Isolation, Durability (ACID) [HR83] – that characterized the new workload type. This workload type waved through almost every area of daily operations (i.e., not only financial services) and is later known as Online Transaction Processing (OLTP) (e.g., explanations by Inmon [Inm05, Pages 4 ff. and 26 ff.].

We argue, a database workload consists of more than queries even though from the outset most database (optimization) research were focused on query optimization[8] (e.g., index selection [HC76], for DSM [KCJ+87]; cf. Section 2.4). For expressive consideration, we have to take into account all database operations (e.g., insert/delete, cursor; cf. TPC-C benchmark[9]) even though single database operations (e.g., transactions) are too transient for specific optimization. In the aggregate, they can have a crucial impact and optimization makes sense.

In the early 1990s, ideas for data retrieval came into focus of research again, which already induced the DSM development in 1970s. However, goal was not only the support of special use cases (e.g., clinical studies, governmental data), but the support of strategic (business) decisions as well as evaluation of businesses performance. That is, requirements for DBSs changed from data retrieval to complex analysis over large data sets within the database. The requirements for such analyses were completely different from requirements for OLTP workloads and Codd et al. formulated the term Online Analytical Processing (OLAP) [CCS93] (cf. also Inmon [Inm05, Pages 175 ff.]). Nowadays, the importance of OLAP is also reflected by development of several of benchmark from the TPC[10] for decision support solely (i.e., TPC-D, TPC-DS, TPC-H, TPC-R, and TPC-DI – for ETL).

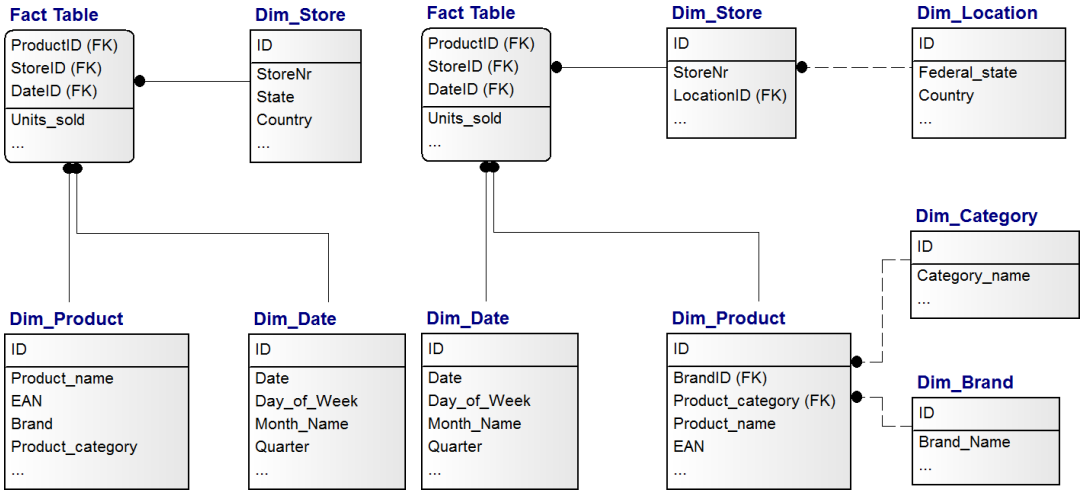
Codd et al. proposed an OLAP server which had access to all data pools (i.e., not only relational DBMSs) and supported different types of analysis based on collected data. However as hardware gets cheaper and cheaper, a special type of DBSs occurred for superior OLAP support – the Data Warehouse (DWH) [Inm05, Pages 29 ff.]. DWHs are no DBSs for daily-operations but they extract and historicize data from any desired source. Therefore, such systems use ETL processes to integrate

[7] ATM – Automated teller machine.

[8] Note, query optimization refers to lookup on (i.e., find) data what always implies a query.

[9] `http://www.tpc.org/tpcc/default.asp`.

[10] `http://www.tpc.org/information/benchmarks.asp`.



(a) Exemplary STAR schema. (b) Exemplary Snowflake schema

Figure 2.5.: *Typical DWH schemas for relational DBMS.*

heterogeneous data. A major benefit of separated DBSs for OLTP and OLAP is the freedom of schema modeling. That is, special schema types evolve over time to support DWHs/OLAP. To the best of our knowledge, widest-spread schema types are the STAR [Inm05, Pages 126 ff.] and the Snowflake schema [Inm05, Pages 360 ff.] (cf. Figure 2.5) respectively. Both schemas have in common that they use a *fact table* as central aspect to represent business objects. The fact table connects all dimension directly (cf. Figure 2.5(a)) respectively indirectly (cf. Figure 2.5(b)) with each other; whereby dimensions qualify attributes (i.e., features) of in the fact table represented business objects.

Nowadays, mixed requirements arise (cf. Chapter 3). Therefore, Cole et al. [CFG+11] proposed the TPC-CH benchmark – a mix of the TPC-H and the TPC-C benchmark. The TPC-CH schema is a modified TPC-C schema (e.g., relations `SUPPLIER`, `NATION`, and `REGION` from TPC-H added). We will use these three benchmarks for our experiments later. Furthermore, we will use the term *workload domain* (in short *domain*) for the application fields: OLTP, OLAP, and mixed OLTP/OLAP that are represented by the three mentioned benchmarks.

2.4. Query Optimization & Self-Tuning Methods

For relational data management, different architectures support different workload domains. Nevertheless, more fine-grained approaches for these architectures exist, which ease computation of tasks and optimize their execution. Therefore, we give an overview of basic approaches for execution and optimization of tasks on which our work is based on. Thus, we discuss principles of query processing, optimization, and their historical development in the following.

Query optimization is a crucial task for DBSs as long as data management exists even as it was based on files in the early days of data management. At this time, the major goal for query optimization was the efficient reduction of DBS respond time. Therefore, researchers proposed different file-based (e.g., index-/key-organized files [SHS05, Pages 140 ff.], heap files [SHS05, Pages 150 ff.]) and tree-based approaches (e.g., tree versus binary search on frequently changing files [Sus63], the B-tree [BM70, BM72]) to find corresponding records (e.g., with a pointer). Such approaches were suitable for unique elements typically the primary key. A comprehensive overview for searching techniques (i.e., locate queried record) can be found in [Knu73, Pages 392 ff.] (or later editions). Nowadays, most DBMSs index primary keys by default to support fast key-record access; whereas the index type depends on implementation guidelines (e.g., B-tree, hash index – challenge to find a hash function [Knu73, Pages 513 ff.]).

For many applications, it was necessary to access secondary keys (i.e., attributes of a record), thus, only indexing of primary keys was not sufficient. Researchers considered this challenge very early. First notable contributions addressed methods and multi-list organization for secondary keys [Joh61, PG63], which influenced the development of inverted files [DL65]. Further research led to a number of other secondary-index structures (e.g., B^+-tree [Knu73, Pages 559 ff.] – nowadays, widest-spread index structure) as well as to retrieval on several attributes (e.g., combined indexes [Lum70], combinatorial hashing [Riv76]). We refer to [Knu73, Pages 559 ff.] for an overview of (historical) searching (i.e., retrieval) on secondary keys.

However, the DSM [CK85] itself (as well as its predecessors; cf. Section 2.2) was in the broadest sense a structure for efficient secondary-key access that primarily had the goal to reduce disk access (i.e., I/O cost). We argue that selection of the optimal decomposition schema (with clustering) tends to be NP-complete just as physical design using index structures [Com78, ISR83, FST88, RS91, CBC93b, CBC93a]. Moreover, Batory showed in [Bat79] that optimal query processing is NP-hard on a given decomposed schema (i.e., transposed files). We suggest, the selection of optimal physical design respectively index candidates is a NP-complete problem in general [Mun57, Cab70, SZ79, CS95]. A comprehensive overview of the NP-problem class can be found in [KPP04]. The decision problem became even more complex due to wider spread of OLTP, thus, different batches (i.e., workloads) were not known in advance anymore (cf. Section 2.3). We discuss the complexity for physical design along with our approach in Section 5.2 more detailed.

Having index structures (i.e., alternate access paths) is not sufficient for optimal query processing. Therefore, index structures have to be used at first, and at second the possible best access path have to be selected for a query respectively operation. Therefore, query-decomposition and access-path-selection approaches were proposed in the 1970s [WY76, SAC$^+$79]. These approaches estimated costs of certain access paths to select the optimal path.

To the present day, DBMSs therefore create and evaluate several query

plans [JK84, Cha98], whereby the optimal query plan[11] is selected. In more detail, we denote the first optimization step as physical optimization (i.e., create possible internal plans with access paths), and the second step, we denote as cost-based optimization that uses statistics to compare internal plans (cf. Figure 2.6). Due to infinite amount of (potential) query plans, cost-based optimization is a very costly task (i.e., towards NP-complete) like index-candidate selection (cf. paragraph above). Moreover, new approaches for index structures (e.g., bitmaps [O'N87, CI98] or join indexes [Val87]) and joins [12] (e.g., hash-joins [DG85], sort-merge join [Gra94a]) increased the search space further. We note, research in the field of join processing is in discussion for decades (e.g., [ME92, GLS94, MBNK04]) as well as research on index structures is not yet finished nowadays [SGS+13, FBK+16].

However, a preceding optimization step is implemented in current DBMSs that overcomes degenerated query-plan search – the logical respectively algebraic optimization. The algebraic optimization [Fre87, GD87] prunes the solution space of (potential) query plans by transformation rules (e.g., for join order respectively type). We highlight, rules for relocation of operation like *move down predicate selection as far as possible (in the query plan)*, for equivalent terms, and for redundant operations were derived from first consideration concerning rule-based query optimization. Nowadays, most (commercial) DBMSs prune the solution space before execution of cost-based query optimization by rule-based optimization [GD87, Fre87, Sel88]. We present an overview for abstract query optimization in relational DBMSs in Figure 2.6 (cf. also [KBL05, Pages 409 ff.] or [Ioa96, Cha98]).

The query-optimization procedure is stable for years due to the fact that most modern query engines (including the optimizer) are Volcano-like [GD87, Gra90, GM93, Gra94b, CG94], which does not mean that there is no research on query processing (e.g., [GHQ95, GPSH02, ENR09]). Volcano-like query engines are extensible for new operators (e.g., new join techniques), evaluate queries in parallel, and support dynamic query plans which remain optimal even if parameters change (e.g., selectivity). Nevertheless, physical design in terms of access structures (e.g., indexes [CDF+01, HIKY12]), specific domains (e.g., DWH design [FBB07, BCB10, BBJB14]), and architectures (e.g., DSM-like approaches [SBKZ08, ABH09] or in-memory processing [Pla09, KN11]) is still very volatile and of high research interest.

An important aspect for physical design is the increasing number requirements (e.g., more aggregate processing [CS95]) that contradict the essential requirements – mostly OLTP – even though research on existing physical-design approaches especially index selection is still an issue [CFM95, CDN04]. Moreover, OLAP becomes more and more important for daily operations; whereby not each task can be externalized to specialized DBS (i.e., DWHs). Therefore, new approaches like materialized views were developed [GHQ95, CKPS95, SDJL96, BDD+98]. Materialized views precompute complex operations like aggregates or joins but suffer from updates due to result materialization.

[11] Note, there is no guarantee to find the optimal plan.
[12] Early systems often just support nested-loop and merge join as in [SAC+79].

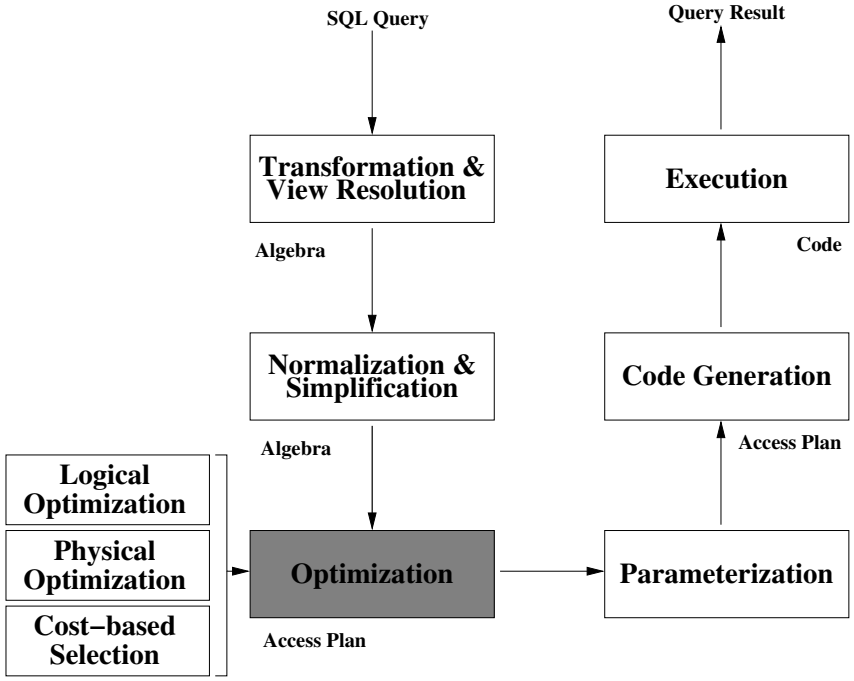
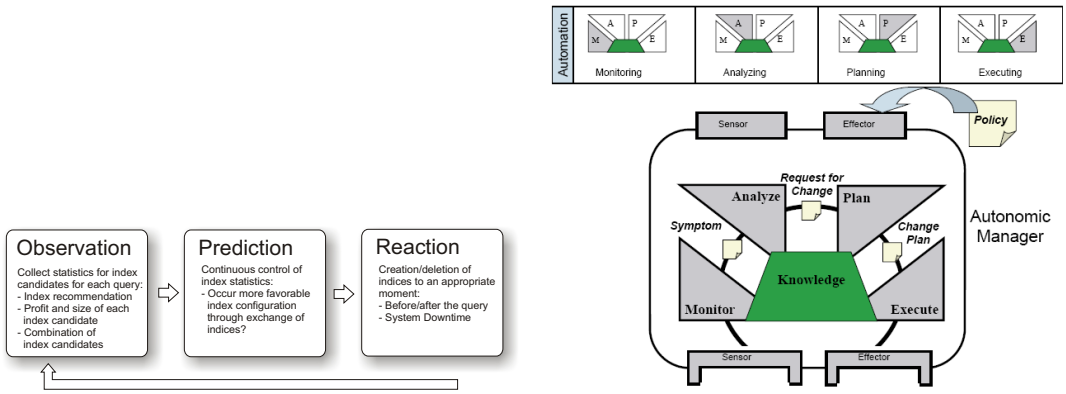


Figure 2.6.: *Abstract overview of query optimization in (relational) DBMSs.*

From 1990s until today, a major challenge for physical design is the efficient utilization of different design approaches for more volatile workload requirements because physical design is static in actual purpose. That is, physical redesign is not applicable for a single query (or small set of) as well as it is a too costly (manual) task to perform in short intervals, thus, such approach is not sufficient to react on changing workloads. Therefore, researchers proposed automatic tuning approaches [WHMZ94, SSV96, CN97, GHRU97, CN98], which support users (e.g., administrators) in tuning physical design of their DBS. That is, such approaches propose a physical design fairly close to optimal (e.g., set of indexes or materialized views) with respect to given constraints (e.g., disk space; cf. Equation 2.1). Such approaches still struggle with multiple-query optimization where queries access same attributes or can share intermediate results – a well-known problem (e.g., by Sellis [Sel88] for query processing). Two major trends occurred in the following years.

First, a number of design advisors were implemented that propose physical design improvements by user's request; respectively they alert user to redesign whenever a benefit above a given threshold is estimated for redesign (e.g., by IBM [VZZ$^+$00, KLS$^+$03, ZZL$^+$04, ZRL$^+$04] or by Microsoft [ACN00, ACK$^+$04, BC06, BC07]).

Second, more sophisticated self-tuning approaches [WKKS99] were proposed by several researchers in various characteristics (e.g., for indexes [SGS03, SSG05, Lüb07, LSSS07b, Lüb08, GK10] or storage management [Ora03a, IBM06c, Ora07, Lüb09]) that have the goal to avoid (or at least to reduce) user interaction for physical design tuning. In this context, materialized views became Automatic Summary Tables



(a) (b)

Figure 2.7.: *Abstract approach for self-tuning databases: (a) according to Weikum et al. [WHMZ94, WKKS99] and (b) by IBM [IBM06a].*

(ASTs) [LGB08, LGB09] which are automatically created, merged, and administered.

We refer to [WMHZ02, CN07, Bru11] for a comprehensive overview for automated physical design and tuning. Two aspects all self-tuning approaches have in common. First, the profit calculates the advantage for an estimated (new) configuration compared with the existing configuration (cf. Equation 2.1). The profit calculation has to satisfy side conditions for system environment (e.g., available size) and (i.e., a threshold – mostly minimum benefit). Second, the strategy for automatic database tuning is in the broadest sense equal – *observe, predict, and react* (cf. Figure 2.7) – whether it is called feedback control loop [WHMZ94, WKKS99] or MAPE [IBM05, IBM06a].

$$
\begin{aligned}
profit(C_{new}) \quad & - \quad profit(C_{current}) > \text{min difference} \\
\text{with subject to:} \quad & max \quad \sum_{I \in C} profit(I) \\
& \quad \sum_{I \in C} size(I) \leq \text{max size} \\
profit(I) \quad & = cost(Q_{current}) - cost(Q_{estimated})
\end{aligned} \tag{2.1}
$$

Finally, we state that even if abstract query-processing procedure and self-tuning approaches are equal for relational DBMSs, specialized approaches exist for either architecture. We highlight special query processing (e.g., different query decomposition approaches for NSM [KY83] or query processing for DSM [KCJ+87]) or architecture-unique index (e.g., cracking – range indexing of columns [IKM07a, IKM07b]) respectively self-tuning (e.g., automatic cracking [Idr10, HIKY12]), but we do not consider these aspects in more detail here.

3. The Dilemma of A-priori Storage-Architecture Selection

Chapter 3 shares material with [LS10, Lüb10, LKS10].

We discussed general knowledge on relational data management, application scenarios, and their optimization (cf. Chapter 2). In this chapter, we discuss the impact of storage architectures to different workload domains (i.e., application fields). Therefore, we discuss features of both (relational) storage architectures (i.e., Column Store and Row Store) and their impact on application fields (cf. Section 3.1). In Section 3.2, we discuss challenges for mixed workload domains (i.e., OLTP/OLAP) and the impact on the design process. Finally, we present a case study (cf. Section 3.3), in which we perform an OLAP benchmark on both architectures, and discuss impacts on query execution on different architectures.

3.1. Read- vs. Write-optimized DBMS

For a discussion on different architectures concerning read- and write optimization, we have to consider correlation between architectures and workload domains in more detail. As discussed in Chapter 2, Row Stores are rather designated for write workloads (i.e., OLTP); whereas Column Stores are more designated for read workloads (i.e., OLAP).

Therefore, we give an overview to different storage types and core features for both architectures (i.e., Column Store and Row Store) in the following that have an impact on domain-specific performance of either architecture. Furthermore, we discuss the increased complexity – due to aggravated performance estimation – for database design and tuning with respect to both architectures.

First, both storage architectures differ in the way, how they partition relations (i.e., tables). Row Stores partition relations horizontally and store these tuple-wise (i.e., all attribute values are sequentially stored for each tuple). In contrast, Column Stores store values of attributes sequentially (i.e., columns; cf. Figure 3.1). Consequently, Column Stores have to reconstruct tuples during the query execution at a certain moment whenever more than one column is affected [HLAM06, ZNB08, MBNK04]. We state, different materialization strategies for Column Stores exist [AMDM07] that differ in point of time tuples are reconstructed. To mention are (a) early and (b) late materialization. For (a) early materialization columns are added to the intermediate result (i.e., stitched together) on access whenever corresponding columns are needed for further processing or final result. For (b) late materialization, columns

are scanned for predicate selection, whereby the result is a position list of attribute values (e.g., a bitmap) that satisfy the predicates. For computation of intermediate results, position lists are intersected (e.g., logical AND) to figure out corresponding attribute values (i.e., alike vectors). In consequence, columns are rescanned for attribute values, which satisfy all predicates, based on the intersected position list[1].

However, we mostly observe aggregations and groupings in the DWH domain (i.e., OLAP workload) that often are processed over single columns. Therefore, Column Stores reduce the overhead for aggregations (e.g., I/O cost) because only affected columns are accessed on. In fact, Column Stores are faster than Row Stores on classic OLAP workloads[2] [SAB+05, HLAM06, AMH08, HD08] without OLTP parts.

We argue that the advantage of vertical partitioning in Column Stores for OLAP is simultaneously a downside for write operations (e.g., updates). That is, tuples have to be reconstructed at first, and subsequently to be partitioned during update operation once again due to column-oriented architecture. In contrast, Row Stores perform better on tuple operations (i.e., OLTP-like operations; e.g., updates) due to tuple-wise access that are not negligible even in the DWH domain. Several approaches [Aba08, Pla09, SBKZ08, VMRC04] try to overcome update problems by Column Stores but none reaches competitive performance compared to Row Stores. In addition, Abadi et al. show that vertical partitioning (of relations) in Row Stores is not a suitable compromise for OLAP [AMH08]. Summarizing, Row Stores show strengths for write and weakness for read workloads, but we argue that Row Stores show competitive results for read workloads to some extent. For Column Stores, the correlation is vice versa, whereby we argue that Column Stores and write operations are not mutually exclusive in general.

Second, Column Store and Row Stores differ in their core features as well. On the one hand, Row Stores utilize indexes and materialized views to improve the performance. Therefore, DBMS vendors and researchers develop a number of self-tuning techniques [CN07] that tune DBMSs automatically. On the other hand, such techniques (e.g., index self-tuning) to the best of our knowledge do not exist for Column Stores in similar manner. Consequently, Row Stores use currently more mature self-tuning frameworks to take action on workload changes [CN07] than Column Stores do.

In contrast, Column Stores have – compared to Row Stores – superior support of compression techniques [AMF06, Aba08]. That is, Column Stores usually support a number of compressions that can optimally be chosen for each column with respect to its data type. Moreover, some Column Stores are able to process compressed data directly [AMF06, Aba08]. Row Stores have to use one compression for a tuple or tuple partition [AMH08] (i.e., the selected compression is always a compromise that needs to satisfy all data types of a tuple).

Additionally, Column Stores and Row Stores use different query-processing tech-

[1] We note, an implementation can be a static order of all columns with respect to a key column.
[2] `http://www.tpc.org/tpch/results/tpch_perf_results.asp`.

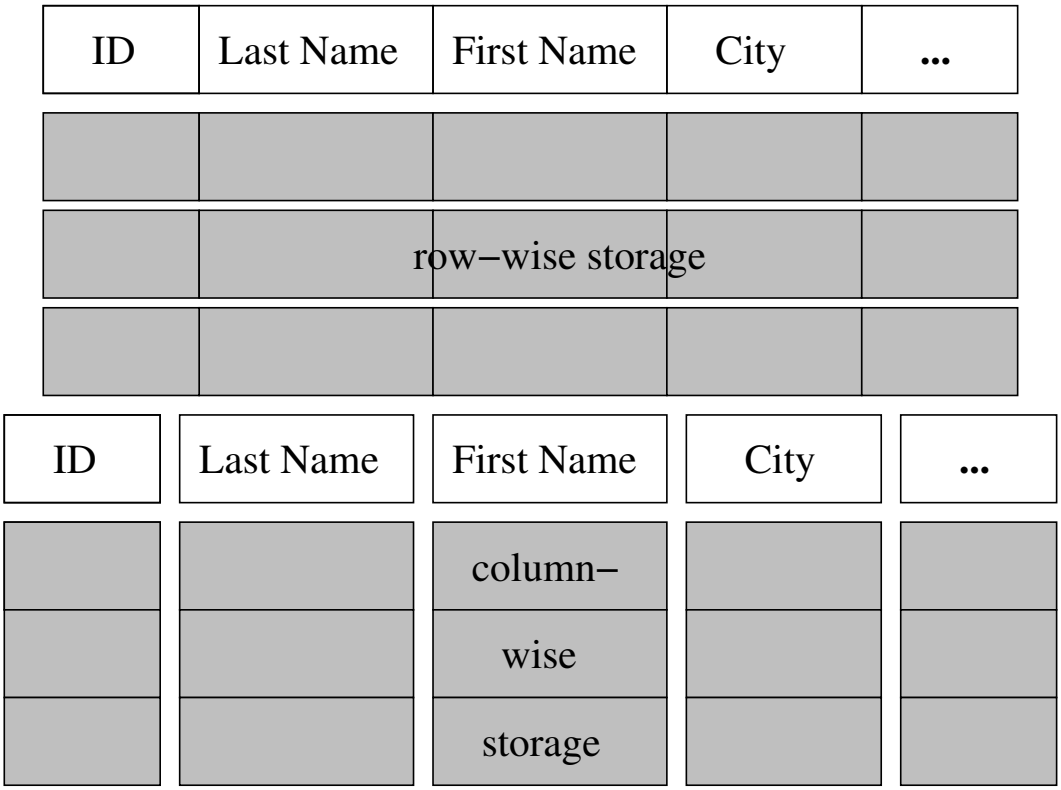


Figure 3.1.: *Storage layout for Row Store and Column Store.*

niques that are caused by different storage architectures and data-flow paths. On the one hand, Column Stores have to reconstruct tuples during query processing, whereby the point of time tuples are reconstructed has crucial impact on query processing [AMDM07]. On the other hand, Row Stores can directly process several columns (without tuple reconstruction), but they always access entire tuples, even if just a single column has to be processed. Finally, Row Store's query processors are always tuple-oriented no matter how data is partitioned. Column Stores can utilize row-oriented as well as a column-oriented query processors [Aba08] (e.g., in correlation to early or late materialization), whereby the performance already affected by query-processor selection.

In conclusion, we argue that complexity of DBS design and its tuning is increased by boosted usages of Column Stores in the DWH domain. Hitherto, we estimate Row Store performance for a given workload and tune DBSs concerning a given workload. We are able to easily compare several systems (i.e., Row Stores) because their core functionality only differs very slightly. Nowadays, we have to estimate DBMS performance across two architectures. That is, we have to select most suitable architecture for a given workload.

One may argue, Column Stores are most suitable for DWH applications in general because they perform better on essential tasks for the DWH domain (e.g.,

aggregations). However, we argue that application fields exist where neither Column Stores nor Row Stores are suitable for each (part of) workload. In line with others [Aba08, Pla09, SB08, SBKZ08, VMRC04], we argue that mixed requirements from OLTP and OLAP arise for modern systems (e.g., updates for (near) real-time DWHs) [ZAL08].

3.2. Challenge to Optimize for OLTP/OLAP Workloads

We discussed storage type and core features of the two major relational architectures – Row Stores and Column Stores – in correlation to the main workload characteristics for OLTP (i.e., tuple operations like updates and point queries) and OLAP (i.e., read workloads with aggregation and groupings). For workloads corresponding to one domain, we figure out that a one to one correlation (i.e., OLTP on Row Stores and OLAP on Column Stores can be feasible and sufficient. However, we suggest that consideration of individual domains is not sufficient nowadays, because new requirements arise in recent years, which imply a mix of workload domains (e.g., real-time update for DWHs [VMRC04, SB08, ZAL08] and real-time analysis [Van01, Lan04, MKKI13, LBH$^+$15]). In consequence, boundaries become obscure between domains, and with that between architectures. Hence, we discuss the challenge of mixed workload-domain optimization (i.e., mixed OLTP/OLAP) in this section.

We argue, new applications demand for efficient OLAP (e.g., aggregations) as well as efficient OLTP processing (e.g., updates) or at least partial support of both domains (e.g., for real-time data or dimension updates). So far, we estimate performance for DBMS (i.e., Row Stores) according to their capabilities on optimization and tuning [Ioa96]; due to negligible differences in their core functionality, they often differ in implementation details only.

Nevertheless, correct performance and progress estimation is very complex and tricky [CKR05] (e.g., often worst-case estimation only). For the described environment, we argue that workload analysis based on queries is suitable because queries are processed in similar manner, thus, a comparison of query-execution times is sufficient with respect to constraints. However, this assumption does not hold whenever a second architecture – that differently processes data – is included to estimation or ranking. Hence, we require new approaches for workload analysis, that are able to compare different DBMSs across architectures (i.e., Row Store and Column Store), which differ significantly in storage, functionality, and query-processing techniques.

We state: To compare Column Store and Row Stores, we have to estimate performance with respect to given workloads and constraints for both architectures. Due to different performance and processing scheme on certain operations for both architectures, we state that workload analysis has to descend to operation level. We argue that such adaptation is necessary, because certain operations (e.g., tuple reconstructions) are not always assessable from a query and its structure. Moreover, some single operations (e.g., tuple reconstructions) have a significant impact on the overall performance of queries. Additionally, some operations (may) only exist for

certain architecture (e.g., tuple reconstruction). Consequently, current workload-analysis approaches, and thus, workload-estimation tools, have to be adapted due to the fact; these approaches only support analyses of entire queries and their structure. That is, operation-specific analyses are not supported yet.

However, optimization and physical design is often based on heuristics. For example, Column Stores are chosen for workloads that contain an amount of aggregations and groupings (like OLAP). We state that system architectures are often constructed (with assumptions and) by heuristics. Similarly, DBMSs are selected by application targets (e.g., a reporting application), thus, heuristics determine system's design. One can call a heuristic that states for OLAP applications: Column Stores are most suitable due to the fact that OLAP includes an amount of aggregations and groupings. However, we cannot generalize this assumption even though Column Stores outperform Row Stores for aggregations in general. If a workload contains a large amount of tuple operations besides typical OLAP operations (e.g., aggregations) then Column Store performance is poor due to tuple reconstructions. We argue, processing scheme and necessary operations are not obviously legible from queries even if these queries are aggregation queries.

Furthermore, Row Stores may compensate poor OLAP performance by efficient tuple processing (e.g., tuple reconstruction is not necessary). That is, we have to analyze query operations themselves to correctly estimate the query performance. We present first insights on the impact of single operations (i.e., here tuple reconstruction) in the following section. With help of workload analysis based on operations, we can obtain (weighted) comparable estimates with respect to different operations even though some operations only exist for certain architecture. Such an approach for workload analyses based on operations across different architectures is a major goal of our work.

We note, the term *query* – if not specified differently – includes besides selection queries also updates, inserts and so on from DML. That is, query characterizes a part of a workload for the remaining thesis.

3.3. Study – OLAP on Different Architectures

In this section, we present a case study that shows the differences in query processing and its performance (cf. Section 3.1) by the example OLAP. Therefore, we perform an OLAP benchmark on a Column Store and a Row Store. We additionally show challenges for optimization on OLAP workloads by this example, and with that highlight the even greater challenge for mixed OLTP/OLAP workloads.

In the following, we present details on setup, environment, and our assumptions concerning this study. Subsequently, we discuss insights into benchmark results, which show that processing schemes, underlying operations, and their impact are not easily legible from queries themselves. Finally, we consider challenges for physical (database) design.

3.3.1. Setup, Environment, & Assumptions

Our test environment is an Ubuntu 9.10 64bit system running on Samsung X65 with a 2.2 GHz dual core processor, 2 GB RAM, and 2 GB swap partition. Furthermore, we use Infobright ICE[3] 3.2.2 and MySQL[4] 5.1.37 for our study.

Thereby, ICE represents Column Stores and MySQL represents Row Stores. Our decision to select these DBMSs is based on two main reasons. First, both DBMSs are freely available, and second, both are relatively similar. On the one hand, freely available systems fortify traceability, reasonability, and repeatability; and on the other hand, systems that originate from same roots are more suitable for comparison. That is, both systems use the common MySQL kernel/management services except that they utilize different storage architectures. Of course, Infobright adds functionality to the underlying MySQL (e.g., another storage manager). Nevertheless, to the best of our knowledge, there are no other DBMSs that utilize different storage architectures, and are as similar as these two.

We conclude that no other DBMSs are more suitable to compare impacts on Column Store and Row Store, even though ICE is focused on DWH applications (i.e., read-only), and MySQL is implemented as generic DBMS that is focused on OLTP as Row Stores usually are. We adjust both DBMSs configurations to guarantee the comparability of the results. That is, both systems run on MySQL-standard configuration. We do not create additional indexes or views for both systems, thus, indexes and views are only created by workload or benchmarks DDL.

We use a standardized OLAP benchmark – TPC-H (2.8.0) [Tra08] – with 1 GB data (i.e., scale factor 1) to exclude unintentional impacts by a poor chosen benchmark setup. We argue, the benchmark is representative for the DWH domain. We state, benchmark data (i.e., 1 GB) does not completely fit into main memory for MySQL-standard configuration (e.g., 16MB key-buffer size). We run two test series concerning the TPC-H benchmark, to show, that application fields for Row Stores still exist in DWH domain. That is, Column Stores do not outperform Row Stores at each query. Moreover, we want to show, that storage-architecture decisions can be easily shifted by changing workloads.

First, we perform a test series with the standard TPC-H benchmark to obtain reference values for both DBMSs. Second, we perform a test series with an adjusted TPC-H benchmark. Therefore, we adjusted the TPC-H benchmark in the following way: We change the number of returned attributes for each query (i.e., in the `SELECT` statement). That is, each query returns results without projection[5] (e.g., `SELECT * FROM table`). Listing 3.2 shows an exemplary adjusted TPC-H query (cf. Listing 3.7 for the original query). We state that we add GROUP BY statements to queries `Q6`, `Q14`, `Q17`, and `Q19` to create valid SQL statements, due to the fact; more attributes than the primary aggregation are processed now. Furthermore, we decide to group these four queries by the same attribute (i.e., each query is extended by `GROUP BY`

[3] `http://www.infobright.org`.
[4] `http://www.mysql.org`.
[5] Projection does not change number of tuples but number of attributes per tuple.

```
1 SELECT *,COUNT(DISTINCT ps_suppkey) AS supplier_cnt
2 FROM partsupp,part
3 WHERE p_partkey = ps_partkey AND p_brand <> 'Brand#51' AND p_type NOT LIKE 'SMALL
    PLATED%' AND p_size IN (3,12,14,45,42,21,13,37) AND ps_suppkey NOT IN (
4     SELECT s_suppkey FROM supplier WHERE s_comment LIKE '%Customer%Complaints%')
5 GROUP BY p_brand,p_type,p_size ORDER BY supplier_cnt DESC,p_brand,p_type,p_size;
```

Figure 3.2.: *Adjusted TPC-H query Q16*

```
1 SELECT SUM(l_extendedprice * l_discount) AS revenue
2 FROM lineitem
3 WHERE l_shipdate >= date '1994-01-01' AND l_shipdate < date '1994-01-01' + interval '1' year
    AND l_discount BETWEEN 0.03 - 0.01 AND 0.03 + 0.01 AND l_quantity < 24;
```

Figure 3.3.: *TPC-H query Q6 [Tra08].*

`l_shipdate`). Moreover, we apply these changes to our test series with the standard TPC-H benchmark to guarantee comparability.

Finally, we argue that we exclude three queries from our test series. First, `Q13` is not executable on MySQL-syntax. Second, we remove `Q18` from test series, because MySQL is not able to finish this query. That is, we abort the execution, because it ran for more than 21 hours. In contrast, the execution time on ICE is only 8 seconds for `Q18`. Third, `Q21` has an extreme high execution time (i.e., 6 hours) on ICE that indicates optimizer problems for this query. MySQL executes `Q21` in 2 minutes and 48 seconds only. We present an overview for both test-series results in Table 3.1 and discuss these in the following section.

3.3.2. Impact of Architectures to Query Execution

We argue, ICE performs better on typical OLAP queries with aggregates on large data sets (e.g., `Q6` – cf. Listing 3.3); whereby ICE's performance is not outstanding in comparison to MySQL for all queries. Our study shows different impacts on the query-execution time of the queries. We observe *three* different impacts in our study.

First, we cannot figure out an impact of our adjustments on the query execution of MySQL – as anticipated. We expect this behavior because Row Stores process

```
1 SELECT n_name,SUM(l_extendedprice * (1 - l_discount)) AS revenue
2 FROM customer,orders,lineitem,supplier,nation,region
3 WHERE c_custkey = o_custkey AND l_orderkey = o_orderkey AND l_suppkey = s_suppkey AND
    c_nationkey = s_nationkey AND s_nationkey = n_nationkey AND n_regionkey = r_regionkey AND
    r_name = 'AMERICA' AND o_orderdate >= date '1994-01-01' AND o_orderdate < date
    '1994-01-01' + interval '1' year
4 GROUP BY n_name ORDER BY revenue DESC;
```

Figure 3.4.: *TPC-H query Q5 [Tra08].*

```
1 SELECT l_returnflag,l_linestatus,SUM(l_quantity) AS sum_qty,SUM(l_extendedprice) AS
    sum_base_price,SUM(l_extendedprice * (1 - l_discount)) AS sum_disc_price,SUM(l_extendedprice *
    (1 - l_discount) * (1 + l_tax)) AS sum_charge,AVG(l_quantity) AS avg_qty,AVG(l_extendedprice) AS
    avg_price,AVG(l_discount) AS avg_disc,COUNT(*) AS count_order
2 FROM lineitem
3 WHERE l_shipdate <= date '1998-12-01' - interval '117' day
4 GROUP BY l_returnflag,l_linestatus ORDER BY l_returnflag,l_linestatus;
```

Figure 3.5.: *TPC-H query Q1 [Tra08].*

	Standard TPC-H		Adjusted TPC-H			Standard TPC-H		Adjusted TPC-H	
#	ICE	MySQL	ICE	MySQL	#	ICE	MySQL	ICE	MySQL
Q1	**00:25**	00:26	**01:18**	00:28	Q11	00:01	00:00	00:22	00:01
Q2	00:45	**01:31**	01:09	**01:34**	Q12	00:02	00:04	01:00	00:04
Q3	**00:03**	00:28	**01:11**	00:27	Q14	00:01	00:32	00:43	00:31
Q4	02:32	00:05	02:42	00:05	Q15	**00:01**	**00:08**	**00:02**	**00:08**
Q5	**00:03**	**01:25**	**01:06**	**01:31**	Q16	**00:01**	00:09	**00:24**	00:12
Q6	00:00	00:03	00:40	00:04	Q17	24:15	00:01	24:41	00:01
Q7	**00:03**	**00:30**	**00:04**	**00:30**	Q19	00:03	00:00	00:31	00:00
Q8	00:02	00:05	00:02	00:05	Q20	10:48	00:01	10:51	00:00
Q9	00:05	00:50	01:09	00:48	Q22	19:21	00:01	19:23	00:01
Q10	00:08	00:10	02:06	00:12					

Table 3.1.: *Comparison of query-execution times (in mm:ss) for ICE and MySQL on TPC-H and adjusted TPC-H.*

inherently (entire) tuples, thus, no drawbacks arise. There are only some queries that show a negligible impact (e.g., `Q2` – cf. Listing A.1, `Q5` – cf. Listing 3.4); whereas the difference is 3 respectively 6 seconds concerning a query-execution time 1 minute. We argue, projections, which prepare final results in a query plan, have no impact on query-execution time for Row Stores because unnecessary attribute will only be cropped from result sets. That is, projections on intermediate results reduce their size but not computational cost (e.g., for join results) due to the fact that the number of tuples to be processed remains. This does not hold for Column Stores.

Second, ICE shows crucial impact for several queries (e.g., `Q1`, `Q3`, `Q5`, and `Q16`). We argue that the largely increased query-execution time for these queries (i.e., for ICE) results from tuple reconstruction on the greatly increased size of tuples (e.g., 6 from 16 for `Q1` respectively 9 to 34 for `Q3`, cf. Listings 3.5, and A.2). Hence, such analyses, which induce very large tuples for (intermediate as well as final) results, crucially worsen performance of Column Stores for OLAP. We conclude that we cannot disregard size of tuples within queries for storage architecture decisions, especially not for analysis and reporting tools that process huge data sets.

Third, some queries do not show an impact according to our adjustments of the TPC-H benchmark (e.g., `Q7` or `Q15`). We argue, tuple-reconstructions costs for these queries do not have a major share of total costs. That is, result sets and the interme-

```
1  CREATE VIEW revenue0 (supplier_no,total_revenue) AS
2      SELECT l_suppkey,SUM(l_extendedprice * (1 - l_discount))
3      FROM lineitem
4      WHERE l_shipdate >= date '1993-05-01' AND l_shipdate < date '1993-05-01' + interval '3'
         month
5      GROUP BY l_suppkey;
6
7  SELECT s_suppkey,s_name,s_address,s_phone,total_revenue
8  FROM supplier,revenue0
9  WHERE s_suppkey = supplier_no AND total_revenue = (
10     SELECT MAX(total_revenue) FROM revenue0)
11 ORDER BY s_suppkey;
12
13 DROP VIEW revenue0;
```

Figure 3.6.: *TPC-H query Q15 [Tra08].*

diate results are comparatively small (e.g., number of involved attributes is largely reduced by the contained view; cf. Listing 3.6). Furthermore, we state that final projections have no impact on execution of `Q7`, because the same projection is done before grouping and aggregation (i.e., `SUM(volume)`; cf. Listing A.4). We conclude that lion's share of costs is caused by other operations for these queries.

We state that queries `Q4`, `Q17`, `Q20` and `Q22` have to be separately considered[6]. We argue, these queries are outliers at least for ICE. Very long query-execution times for these queries on ICE indicate to the same issue with respect to their query structure (i.e., all these queries are nested; cf. Appendix A.1). We assume that ICE optimizer or ICE query processor causes an issue while processing (complex) nested queries on large relations (i.e., `CUSTOMER` or `LINEITEM` – the fact table). However, our results for these queries also show that there is only a negligible impact on both systems by our adjustments, due to the fact, complex joins cause ICE to reconstruct a large number of tuples, whether with or without projection, whereas MySQL processes inherently (entire) tuples without additional cost.

Consequently, we argue that we cannot easily figure out general decision rules for query optimization across architectures based on the query structure (e.g., SQL syntax). Already for OLAP, the consideration is very complex and not definite. That is, we have to analyze impact on single operations to total query costs (e.g., joins, tuple reconstruction). This also holds for optimal storage-architecture selection for a given workload (or at least a query), because different impact of operations is not obvious by query structure or syntax. In other words, akin queries[7] (e.g., `Q15` and `Q16`; cf. Listings 3.6 and 3.7) cause different impact (e.g., by change of projections). Changing the projection (i.e., a single operation) alters size of intermediate and final results concerning the number of involved attributes, and causes different impacts on several queries.

[6] We note, `Q21` fits into same pattern.

[7] We note, similarity of queries correlates to the existence of simple equality join, aggregates, tuple operation (i.e., `ORDER BY`), and low number of attributes for final result (i.e., ≤ 5 attributes).

```
1 SELECT p_brand,p_type,p_size,COUNT(DISTINCT ps_suppkey) AS supplier_cnt
2 FROM partsupp,part
3 WHERE p_partkey=ps_partkey AND p_brand<>'Brand#51' AND p_type NOT LIKE 'SMALL
    PLATED%' AND p_size IN (3, 12, 14, 45, 42, 21, 13, 37) AND ps_suppkey NOT IN (
4     SELECT s_suppkey FROM supplier WHERE s_comment LIKE '%Customer%Complaints%')
5 GROUP BY p_brand,p_type,p_size ORDER BY supplier_cnt DESC,p_brand,p_type, p_size;
```

Figure 3.7.: *TPC-H query Q16 [Tra08].*

Furthermore, we argue that our assumptions for mutual behavior are confirmed. That is, Column Stores do not outperform Row Stores for each query in OLAP environments. We further argue, there are application fields for Row Store and Column Stores in the DWH domain particularly with regard to mixed workloads (cf. Section 3.2). For mixed OLTP/OLAP workloads, we assume that considerations for query optimization and physical design become even more complex and obscure than for OLAP. We note, we present queries in Appendix A.1 that are not discussed in detail here.

3.4. Summary

In this chapter, we discussed impacts of different (relational) storage architectures (i.e., Column Store and Row Store) on application domains and challenges for physical (database) design on mixed OLTP/OLAP workloads. Therefore, we presented major the distinctions between architectures and emphasized the increased complexity for design decisions. We further considered current optimization approaches; whereby we argue, we had to adapt query-wise to operation-wise analysis to be sufficient for estimation and optimization across architectures. That is, we stated that the impact of single operations (e.g., tuple reconstruction) is crucial for overall query processing or at least overall query performance. In consequence, we presented a study to show first insights into operation impact and discussed the results based on queries from the TPC-H benchmark.

4. Workload Decomposition & Representation

Chapter 4 shares material
with [LGB08, LGB09, Lüb09, LKS10, LKS11c, LKS11b].

If we want to select the optimal storage architecture concerning given workloads, we will have to analyze these workloads. Therefore, we need workload-statistic representations that allow aggregation, processing, as well as administration of extracted statistics. In this chapter, we introduce our approach to decompose and represent workloads independently from the storage architecture. First, we present our architecture-independent approach based on the relational data model [Cod70, Cod72, ABC+76] using existing DBMS functionality. We gather workload statistics directly from existing systems or use samples to process. Second, we introduce our workload-pattern approach. That is, we show details of workload decomposition and discuss different granularities for workload representation. Third, we present the mapping of decomposed query parts to our workload patterns. Fourth, we show a proof of concept. Our approach is applicable to each relational DBMS. Nevertheless, we decide to use a closed source system for the proof of concept because the richness of detail of optimizer output and query plans is higher and easier to understand. We state that more detailed information from optimizers results in more accurate recommendations. Finally, we discuss possible solutions to homogenize statistics from different DBMSs. In Figure 4.6, we illustrate the procedure of our decision process from given workload up to storage-architecture selection.

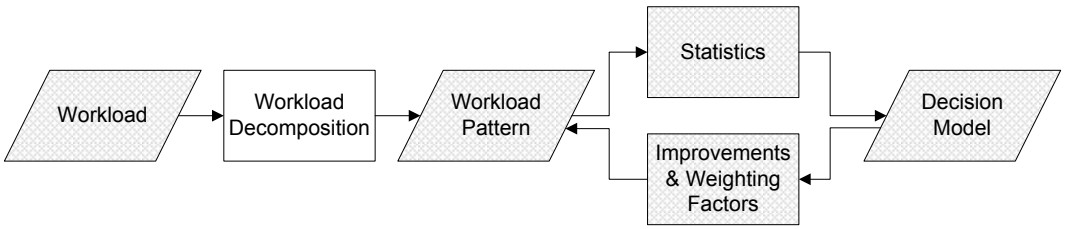


Figure 4.1.: *Overall workflow of the storage-architecture-decision process and chronological classification for the following decomposition approach.*

```
1 SELECT *
2 FROM employees e JOIN departments d
3 ON e.department_id=d.department_id
4 ORDER BY last_name;
```

Figure 4.2.: *Example SQL query (14-1) [Ora10a].*

4.1. Relational Algebra & Query Plans

Different application domains require different storage and optimization approaches (cf. Chapter 3). Our goal is a minimally-invasive and architecture-independent approach in which we process from workload toward decision different job steps (cf. Figure 4.1 for workflow classification). Therefore, we use query plans [ABC+76] that exist in each relational DBMS. On the one hand, we reuse database functionality and avoid new computation cost for optimization. On the other hand, we make use of system optimizer estimates that are necessary for physical database design [FST88]. In line with Finkelstein et al. [FST88], we state that native cost estimates allow efficient physical design for a certain purpose. In the field of physical design, approaches based on native cost estimates lead to several design advisors for commercial DBMSs (e.g., for IBM DB2 [KLS+03, ZRL+04, ZZL+04] or Microsoft SQL Server [BC06, BC07]).

Following established approaches based on cost estimates, we collect statistics directly from a DBMS based on query plans, thus, we use native optimizer-cost estimates. We show an example SQL query in Figure 4.2, its query plan[1] in Figure 4.3, and the corresponding optimizer output (textual query plan) in Table 4.1 [Ora10a] for an established commercial DBMS. Table 4.1 already presents some statistics such as number of rows, accessed bytes by the operation, or cost – an artificial value computed from estimated CPU and I/O cost estimations. Nevertheless, Table 4.1 shows only an excerpt of gathered statistics. All available values for query plan statistics can be found in the Oracle 11gR2 documentation [Ora10d, Chapter 12.10]. Hence, we are able to determine the performance of operations on a certain architecture by statistics – in our example a Row Store – such as CPU cost or I/O cost. We obtain query plans directly from the DBMS optimizer (e.g., by EXPLAIN PLAN) or use sample workloads with distribution of operation-types and their corresponding cost. Hence, our approach is applicable for queries. With respect to our definition of workloads (cf. Section 2.3), we conclude that our approach is also applicable for workloads.

In addition to performance evaluation by several estimated cost, we gather further statistics from query plans which influence performance of an operation on a certain architecture (e.g., cardinality of attributes). For Column Stores, cardinality indirectly affects performance of operations if the operation processes several columns,

[1] Note, $\omega_<$ (sort/order ascending) is not part of the minimal relational algebra that we refer below. We show the operator to be equivalent in our example's representation.

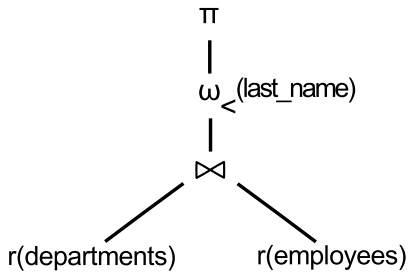


Figure 4.3.: *Query plan of SQL example (14-1) [Ora10a].*

ID	Operation	Name	Rows	Bytes	Cost (%CPU)	...
0	SELECT STATEMENT		106	9328	7 (29)	...
1	SORT ORDER BY		106	9328	7 (29)	...
* 2	HASH JOIN		106	9328	6 (17)	...
3	TABLE ACCESS FULL	DEPARTMENTS	27	540	2 (0)	...
4	TABLE ACCESS FULL	EMPLOYEES	107	7276	3 (0)	...

Table 4.1.: *Textual query plan of SQL example (14-1) [Ora10a].*

thus, Column Stores have to process a number of tuple reconstructions (e.g., high cardinality leads to many reconstructions). Consequently, we use meta-data (e.g., computation of selectivity for attributes) to estimate the influence of data on the performance of architectures.

Our statistic-gathering approach is applicable for relational DBMSs because all relational DBMSs are based on the relational data model and implements database operations to compute on relations. We state that internal database-operation implementation can be reduced to or at least encapsulated in relational algebra operators. Such encapsulation of implementation details to relational algebra is also used by DBMS optimizer (and required) for algebraic query optimization. We conclude, our cost-estimation approach combines different DBMS implementations as well as different storage architectures, thus, we are able to compute the most efficient physical design based on native cost estimations.

4.2. From Query Graph Model to Architecture-independent Workload Representation

In previous work [Bub07, LGB08, LGB09], we developed an approach to analyze, represent, and aggregate given workloads to recommend efficient AST configurations. Similar approaches based on ASTs can be found in [CKPS95, GHQ95, SDJL96, BDD+98, MQM97, GHRU97] based on the Query Graph Model (QGM) [BR91, ZCL+00]. The distinctions are discussed in [ZCL+00]. We evaluate our approach [LGB08, LGB09] with the TPC-H Benchmark [Tra08] and compare the results with IBM DB2 Design Advisor (version 9.1) [IBM06b]. We figure out that our approach [Bub07, LGB08, LGB09] causes a minimal overhead due optimized graph

modification and restructure operations that are competitive with tree operations, e.g., traverse:

$$O\left((\#\text{patterns} \in WG) \times \log_{(\text{order of } T)}(\#\text{patterns} \in T)\right)$$

$$\textit{with } WG \mathrel{\widehat{=}} \text{ Workload Graph } \textit{and } T \mathrel{\widehat{=}} \text{Subtree of } WG.$$

For our framework, we adopt our approach, which recommended AST configurations, and assume that $T \in WG$ has a root and a number of child nodes. Child nodes exactly have one parent node and contain workload statistics. Two equivalent subtrees T and T' can exist in WG that are connected at their root (or parent) node. That is, we aggregate statistics of T and T' if their structure in WG is equivalent (e.g., we aggregate cost of one database operation per DBMS for a given workload). Consequently, T is always a tree structure. For an architecture-independent workload representation, we combine the ideas of query optimization by query plans and our workload-statistic framework to recommend ASTs. Therefore, we take advantage of abstraction in the relational data model which implies implementation of same abstract operators in each relational DBMS. However, different relational DBMSs of the same architecture use different optimization techniques for query execution (e.g., different index types); whereas basic data-flow paths are almost equal (e.g., tuple-wise). Different relational architectures additionally differ in storage type, data-flow path, and optimization techniques for query execution, but still implement the same abstract operators. Consequently, we abstract from different optimization techniques, data-flow paths, and storage types in our framework. Moreover, we state that architecture independence also covers DBMS independence in the field of relational DBMSs. Concerning Section 4.1, we are able to extract native cost estimates using query plans from DBMSs. Further, we observe three additional steps to represent workload statistics architecture independent. First, we decompose query plans to single database operations. Second, we map database operations to workload patterns. Third, workload statistics are stored in our workload graph, thus, we can administer, analyze, and aggregate statistics.

4.2.1. Decomposition to Single Database Operations

We introduce our approach based on the relational data model [KBL05, Pages 35 ff.] and algebra [KBL05, 127 ff.] – the idea suggests itself to decompose query plans based on the basic relational algebra operators. Summarized, we have the following relational operators:

1. the Projection π[2],
2. the Selection σ,
3. the Union $\cup$,
4. the Set Difference $-$, and

[2]Note, Projections do not eliminate duplicates implicitly – as Codd proposed in [Cod70].

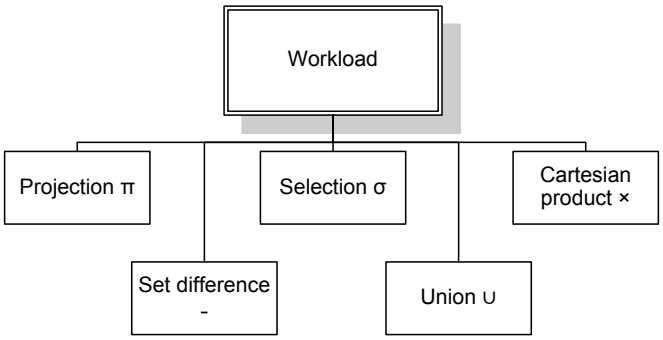


Figure 4.4.: *Minimal set of relational algebra operators in workloads.*

5. the Cartesian product $\times$.

For a closer view to relational algebra operators and derived operators cf. Section 2.1. We state that a workload based on relational algebra is composed of or can be reduced to the above mentioned basic relational algebra operators. Consequently, workloads in relational DBMSs can be decomposed to only five elements (cf. Figure 4.4).

On closer examination, we figure out two different behaviors. First, the Cartesian product as well as derived join operators merge sets of attributes and match tuples of attribute sets to a certain condition (e.g., concatenate each tuple of set A to each tuple of set B (Cartesian) or concatenate tuples only on equivalent join-attribute values (equi-join)). Second, projection, selection, union, and set difference modify tuples of the relation directly; that is, relation structure does not change by these operators. The projection shows a subset of relation attributes whereas the selection shows a subset of relation tuples while the base relation remains unchanged. The union and set difference modify the number of relation tuples only on the condition that relation structure and attribute names are equival (cf. Section 2.1). We conclude that workloads based on the relational algebra have to be decomposed to at least two patterns. The first pattern *concatenate relations* covers the Cartesian product and the derived join operators; the second pattern *tuple operators* covers the modification of tuples in number and depiction of a relation. We present the graphical representation in Figure 4.5.

In line with others [KBL05, Pages 147 ff.], we state that the relational algebra is not powerful enough to support common SQL syntax completely (e.g., SQL:1992 [Int92, MS92, DD97], SQL:1999 [Int99, GP99, EM99], cf. Section 2.1). We can argue in diverse directions but the support of relational algebra extensions is highly dependent on used DBMS. That is, we only depict two general issues. First, SQL is based on multi-sets instead of sets; and second, the basic relational algebra does not support aggregation and groupings. However, we focus on the SQL:1992 and the SQL:1999 standard because we assume that these are most commonly used on the one hand; and on the other hand, newer SQL standards (since SQL:2003) mostly consider object-relational extensions which we do not yet consider (e.g., Java [Int03a] and XML [Int03b]).

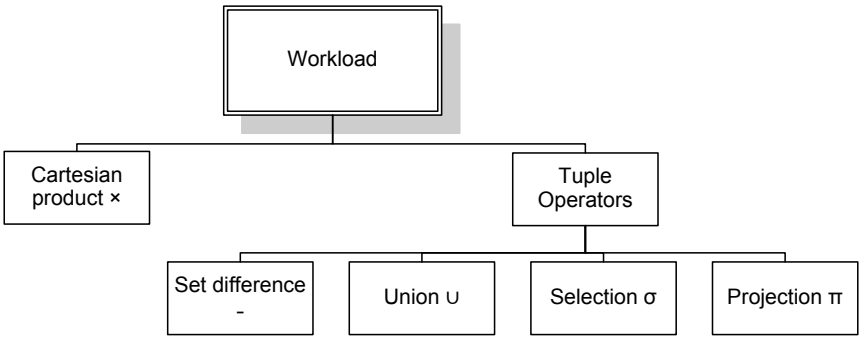


Figure 4.5.: *Decomposed workload based on the five basic relational algebra operators.*

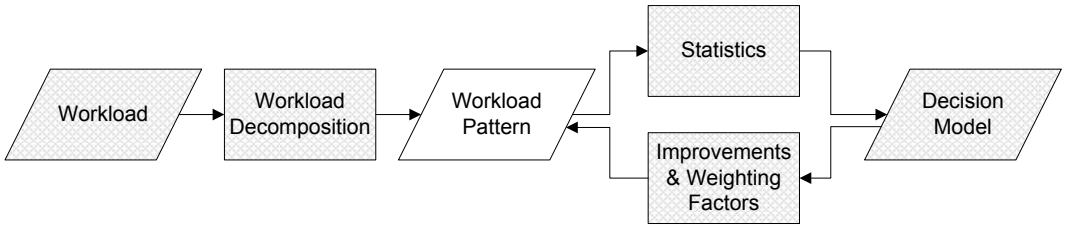


Figure 4.6.: *Workflow – Chronological classification for the following workload-representation approach.*

4.2.2. Map Database Operations to Patterns

We extend our approach to support the power of SQL and to close the gap between our algebra-based approach and SQL; due to the fact that SQL is the universal query language for RDBMS (cf. Figure 4.6 for workflow classification). In Section 2.1, we show that relational algebra and SQL are not equivalent [KBL05, Pages 147 ff.] but mapping approaches exist for systems which are characterized by SQL [CG85, RKB87]. However, our approach already covers two basic operation groups of SQL. First, the concatenate-relations pattern covers the Cartesian product and all derivable join operations (e.g., equi-join and outer joins). We refer to this pattern as *join pattern* in the following. Second, the tuple-operators[3] pattern encapsulates the tuple processing on tables (select rows as well as columns). Nevertheless, we miss another important functionality group of SQL: Aggregation of data. That is, we have to add a third pattern that covers aggregation and groupings from the SQL functionality [MS92] (cf. Section 2.1). Consequently, we obtain the following *three* workload patterns: 1) the ***join pattern***, 2) the ***tuple-operation pattern***, and 3) ***aggregation & grouping pattern***.

We identify different processing schemes (e.g., process columns or rows, concatenate relations) with our patterns. We argue that the performance of operations as well as their impact on the query performance is highly dependent on input and out-

[3]In the following, we refer to tuple operations when it comes to SQL, thus, we distinguish them from tuple operators for the relational algebra.

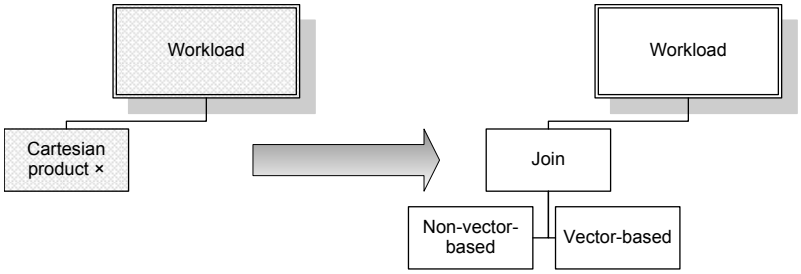


Figure 4.7.: *Derive join pattern from Cartesian product.*

put of operations (cf. Chapter 3). Hence, we have to observe the different processing schemes in more detail. Therefore, we define a number of sub-patterns for each of those three patterns to characterize particular operations more precisely within our workload patterns. That is, we extend our approach from a relational-algebra representation to a representation for SQL-minted systems. We abstract from implementation details in DBMSs and only consider universal operations that modify rows, columns, or tables[4] for query results. However, we enable analyses based on the three patterns and additionally more fine granular analyses based on sub-patterns with our workload-representation approach. That is, we can determine which operations cause the majority of costs within a pattern. In the following, we introduce the sub-patterns that are assigned to one of those three patterns.

From Cartesian Product toward Join Pattern

First, we define the ***join pattern*** based on the Cartesian product[5] to cover operations for the concatenation of relations (joins) of a workload (cf. also Figure 4.7). Join operations are basic within the relational data model. Hence, these operations affect each relational DBMS. However, joins are costly tasks and can affect performance for DBMSs significantly. We determine this pattern to highlight differences between join techniques of Column Stores and Row Stores (e.g., process joins directly on compressed columns or bitmaps). Within this pattern, we distinguish different processing schemes for the concatenation of relations. That is, we do not distinguish between different non-optimized and optimized join implementations (e.g., nested loop vs. merge join), but we distinguish between join processing over tuples and columns. We abstract from data compression in this pattern due to the fact that join techniques process (i.e., concatenate) data and do not consider shapes of data like compression. However, bitmap representation already is a type of compression. We abstract from further segmentation into sub-patterns because bitmap encoding is applicable for Row Stores and Column Stores. We consider the general effects of data compression in another pattern. Consequently, we identify the following two sub-patterns:

[4]We refer to tables in consideration of SQL and to relations according to the relational algebra.
[5]Cartesian product is the first of five basic relational algebra operators that we have to represent.

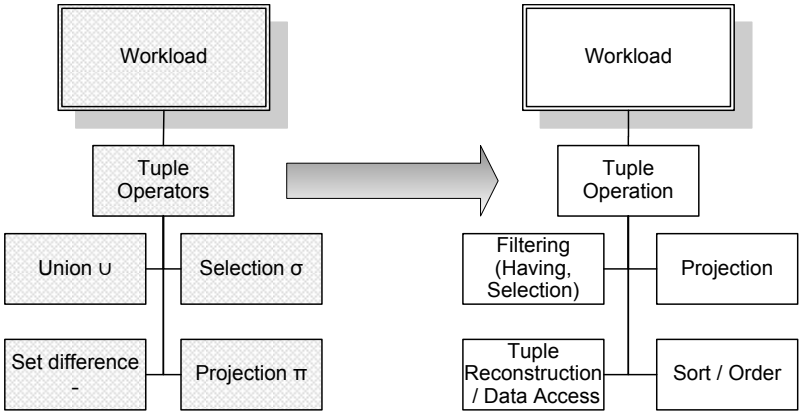


Figure 4.8.: *Redefine tuple operators for SQL-minted systems.*

Vector-based: The column-oriented architecture inherently supports vector-based join techniques (e.g., invisible join [AMH08]) by the present column-wise partitioning. In contrast, Row Stores have to create and maintain structures (e.g., bitmap (join) indexes [SAB$^+$05, Lüb08]), which induce additional costs. We can observe the impact of additional structures to the join performance in general.

Non-vector-based: This sub-pattern represents non-masked (classic) join techniques [ME92] (e.g., sort-merge-join [Gra94a]) to distinguish the performance from vector-based join techniques. Moreover, whenever join performance is influenced by architecture in general, we can observe this behavior.

We identify two sub-patterns only that represent two fundamental different processing schemes of Column Stores and Row Stores. Both sub-patterns cover the representation of the Cartesian product and its derived algorithms to concatenate relations as first (of five) base relational operator in our framework. Different join concepts (e.g., merge or nested loop join) are not represented because they are applicable for both architectures. Hence, we conclude that there is no necessity to map each join concept into separate sub-patterns. As a result, we can estimate effects of architectures to the join performance. Figure 4.7 shows the redefined Cartesian product pattern that we designate as `join pattern` in the following and the associated sub-patterns `vector-based` and `non-vector-based`. Figure 4.10 (on Page 41) shows the final result of our transformation.

Definition of the Tuple-Operation Pattern

Second, we refine the tuple-operators pattern (cf. Figure 4.5) that we refer as ***tuple-operation pattern*** in the following. This pattern represents all operations (cf. Figure 4.8) that modify tuples of a relation in number and depiction (e.g., selection). We summarize them in one pattern to evaluate the performance of tuple processing

(schemes) for each architecture, because Row Stores process directly on tuples in contrast to Column Stores, which have to compute tuple reconstructions to process on tuples.

Therefore, we define the projection sub-pattern and the selection sub-pattern to represent the second and third (of five) basic relational algebra operators in our workload framework. We argue, selection of tuples is not only limited to the selection operator in SQL-minted systems, thus, we designate the selection sub-pattern as filtering sub-pattern in the following. Furthermore, we consider intermediate results in this pattern. We argue that intermediate results are subsets of base relation sets (cf. Section 2.1). Consequently, we also represent the set difference and the union operator in this pattern because both relational operators modify the number of tuples in relations (e.g., as intermediate result).[6]

We argue, set difference utilizes two sets of tuples to select certain tuples from either set of tuples (cf. Section 2.1), thus, we associate the set difference to the filtering sub-pattern. In the same manner as the set difference, the union operator utilizes two sets of tuples. Entirely, we concatenate these sets to one set of tuples with equivalent arity instead of filtering the sets (cf. Section 2.1). That is, we have to access the sets (i.e., relations or their tuples) before we are able to process any of the base operators.

Preceded data access for query processing is necessary for the relational base operators as well as for the following extensions to support SQL-minted systems. Hence, we define the data-access sub-pattern to figure out the accessed amount of data for operations and to consider the data flow within queries. We also associate the union to the data-access sub-pattern due to the fact that we do not perform additional actions to the concatenation of sets. However, we already represent all (five) basic relational algebra operators and the data-flow mapping for these operators in our framework.

In addition, we assign the tuple reconstruction (for Column Stores) to the data-access sub-pattern. In this way, we represent the column-store-specific operation during query processing to materialize and access tuples on the one hand, and on the other hand, we are able to analyze the specific access behavior in contrast to Row Stores. Furthermore, we add the sort/order sub-pattern to support tuple processing on SQL-minted systems, which consider, in contrast to the relational algebra, the order of tuples. In consequence, we identify the following four sub-patterns:

Sort/Order operation: Sort and order operations create certain sequences of tuples and affect all attribute values of a tuple. We assume that duplicate elimination is also a kind of sort operations (e.g., DISTINCT-statement) because at least an internal sort is necessary to find duplicates efficiently. We add this sub-pattern to represent order of tuples and multi-set semantic (duplicates) from SQL-minted systems, because neither multi-set semantic nor duplicates exist in the relational algebra.

[6] We state that the arity of tuples have to be equivalent to process set difference and union.

Projection: Projection returns a subset of attributes of relations and causes (normally) no additional costs. In Row Stores, projection only reduces the arity of tuples by omitting attributes, thus, only storage size of intermediate results can be reduced. In Column Stores, projections determine the number of attributes that have to be reconstructed to tuples. That is, the costs for tuple reconstructions are influenced. Furthermore, this sub-pattern represents the data flow within queries due to the fact that the projection depicts final query results. That is, we represent size of final results in this pattern.

Data access and tuple reconstruction: We map different data-access schemes for Column Stores and Row Stores (i.e., column- vs. tuple-wise) in this pattern. The access-data sub-pattern represents the amount of data and its costs that need to be read from base relations (e.g., from disk into buffer) for processing of this data. Furthermore, we represent the data flow within queries. That is, we map the data flow from the base relations (with this sub-pattern) via data processed by operations (each sub-pattern) to the representation of final results by the projection sub-pattern. In contrast to Row Stores, which access the data tuple-wise, Column Stores have to undo the column-wise partitioning of data (i.e., reconstruct tuples) for the presentation of final results at some point during query processing. Except for access on majority of table columns, column-wise data access reduces I/O, but tuple-reconstruction costs are directly related to column-wise data access. This behavior is independent from materialization strategies, thus, the materialization strategy only effects the degree of freedom for optimization which we do not consider in our workload-representation framework. Consequently, we also represent the tuple reconstruction as column-store-specific operation in this sub-pattern to map the processing of tuples for Column Stores.

Filtering: The filtering sub-pattern covers the relational selection operator (i.e., selection of tuples in relations or intermediate results) that we commonly represent in SQL-minted systems within the WHERE-clause. Furthermore, we represent the set difference (in SQL MINUS or EXCEPT) within this sub-pattern because we select tuples out of two sets which are also represented as relations or intermediate results in DBMSs. However, we represent special filter operations from SQL-minted systems, too. We argue that the HAVING-clause process selection of tuples on special intermediate results named groups, which we thoroughly address in the last workload pattern. That is, we assign the filtering within groups (i.e., with HAVING-statement) to the filtering sub-pattern.

We conclude, we support all five basic relational algebra operators with our framework at this point. We already represent the Cartesian product by the join pattern. So, we define the `tuple-operation pattern` that represents all relation-modifying operators[7] from the relational algebra. The associated sub-pattern `projection` maps

[7] We remark that intermediate results themselves are relations.

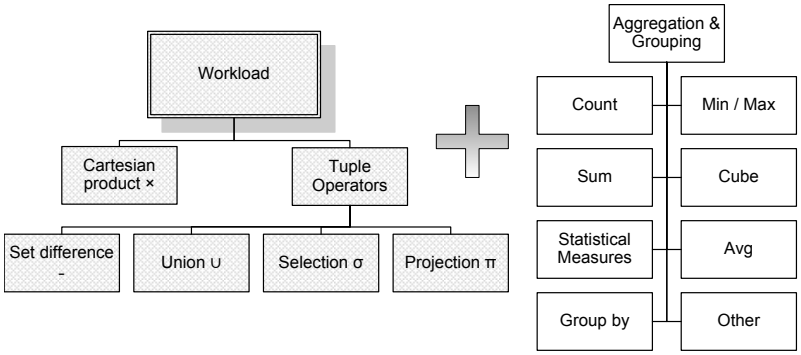


Figure 4.9.: *Add new operations to support SQL-minted systems.*

the arity modification of tuples and the `filtering` sub-pattern maps the relational selection. We entirely have to represent relational set operations. Hence, we assign the set difference (i.e., tuple selection from two sets of tuples) to the filtering sub-pattern and the union (concatenation of tuple sets) to the `data-access` sub-pattern.

We support SQL-minted systems with the `sort/order` sub-pattern that consider the multi-set semantics of SQL, duplicate elimination, and the order of tuples. We further argue that other predicate selections from SQL syntax (e.g., HAVING) are also represented by the filtering sub-pattern. We are able to observe different data-access schemes for Column Stores and Row Stores with the data-access sub-pattern. However, we represent the tuple reconstruction within the data-access sub-pattern to analyze the additional costs for this operation in Column Stores. We show the result in Figure 4.10. Additionally to the complete representation of relational operators and our extensions for SQL-minted systems, we map the entire data flow of queries from base relations (data access) to final results (projection) in our workload-representation framework.

New Aggregation & Grouping Pattern

Third, we add a new pattern group for operations that exist in SQL-minted systems but not in the relational algebra (cf. Figure 4.9). We summarize these operations to one group because these operations process a single column (e.g., average computation) or entirely process up to a small number of columns (groupings). We state that these operations aggregate columns to an expressive value or group equal values of a column in an order of columns. Hence, we name the pattern ***aggregation & grouping pattern*** in the following. We complete the support of SQL-minted systems for our workload-representation framework with this pattern group. We determine aggregation & grouping pattern as counterpart to the tuple-operation pattern. The tuple-operation pattern reflects the Row Stores strengths due to the tuple-wise access. In contrast, the operations grouped to the aggregation & grouping pattern process only a single column or at least a very limited number of columns (e.g., GROUP BY). That is, Column Stores commonly perform well on aggregations and

groupings (cf. Chapter 3). For aggregation & grouping pattern, we identify inspired by the SQL syntax the following eight sub-patterns:

Count operation: The COUNT operation counts the number of attribute values (except NULL) in a column as well as COUNT(*) counts only the number of key values/rows, thus, this operation always processes a single column. We argue, the DISTINCT-statement (cf. Sort/Order pattern) only eliminates multiple-occurring attribute values from standard COUNT-computation. However, we are aware that many COUNT-operations are not computed but read from database statistics.

Min/Max operation: The MIN/MAX-operation computes the minimum respectively the maximum value from all attribute values of a column. These values are often part of database statistics, too.

Sum operation: This operation computes the sum of all (numeric) attribute values according to one selected column.

Average computation: The average computation (mean) processes all values of a single column like the sum operation. Additionally to the sum computation, the COUNT operation counts the number of attribute values to calculate the mean (AVG).

Group by operation: This operation groups unique values of a column according to an order of columns and specifies a subset of relation's tuples which as well can be an intermediate result. Groupings are possible from one to n columns whereas n equals the maximum number of columns of a relation. We state, groupings are computed on a small subset of the maximum number of columns commonly. We already assigned the selection of grouping tuples by HAVING to the filtering sub-pattern (i.e., tuple operation) due to the fact that grouping tuples have to be reconstructed before. We have to reconstruct tuples within groups because single columns are not independent from each other anymore.

Cube operations: The cube operation computes all feasible combination of aggregates for selected dimensions, thus, we argue, the cube operation is a special multidimensional group operation [GBLP95, GCB$^+$97]. The cube computation requires the power set of aggregating columns. That is, n attributes in a cube are computed by 2^n GROUP BY-clauses. A common abbreviated syntax is GROUP BY CUBE that is standardized in the SQL:1999 standard [GP99] as CUBE()-operation. In addition, we consider the ROLLUP operation (also from SQL:1999 standard [GP99]) that process similar to the cube computation [GCB$^+$97] as well as other cube operations (e.g., DRILL DOWN). In contrast to the cube, the ROLLUP computes a specified permutation of attributes and not all permutations. That is, we map most OLAP operations [GCB$^+$97, GP99] to this sub-pattern.

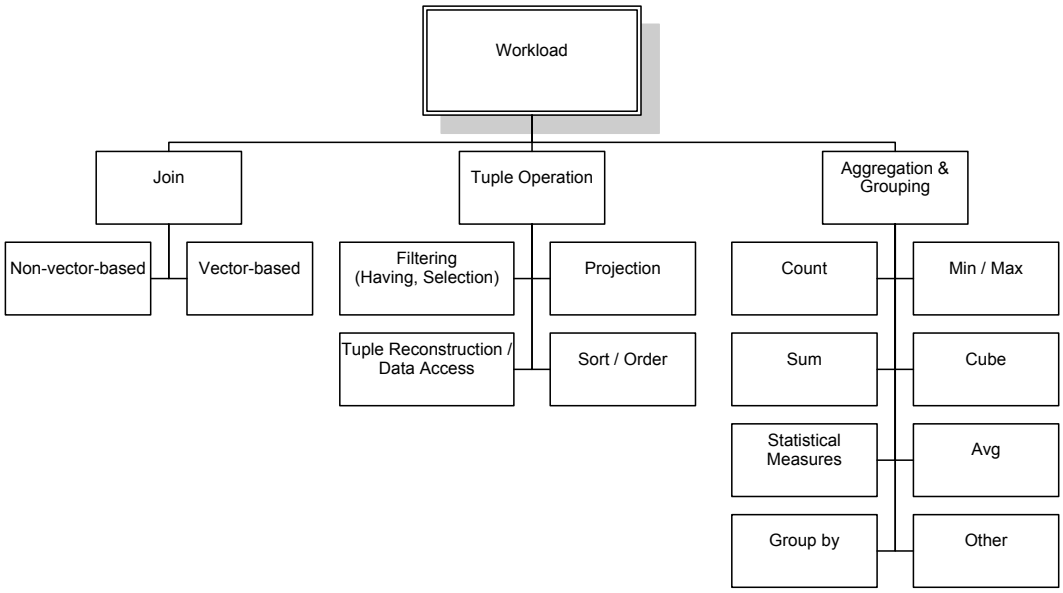


Figure 4.10.: *Workload patterns based on operations.*

Statistical measures: This sub-pattern represents statistical measures (e.g., confidence interval or median) that are processed by SQL-extending functions within workloads. The standard deviation (STD) is a statistical measure for the variability of a data set and is computed by a two pass algorithm which means two cycles. We state that the STD-statement is a good representative for these measures because it is previously implemented in many DBMSs as function (e.g., in PL/SQL [Des07] or Transact-SQL [Hen00]) but is not part of the SQL standards [MS92, DD97, GP99, EM99].

Other: We add this pattern as representative of new processing schemes (i.e., pattern) in SQL-minted systems. In this pattern, we can summarize object-relational aggregations or any other aggregation type in SQL-trimmed workloads. We state that we are able to extend any other pattern with such an additional sub-pattern to support new functionality.

In the third pattern, we summarize aggregation and grouping functions from most common used SQL standards [Int92] namely `count`, `min/max`, `sum`, `average`, and `group by`. Furthermore, we add sub-pattern for important statistical computations and validation (e.g., standard deviation) that we title as `statistical measures`, and map the important OLAP operations on cubes to a separate sub-pattern named `cube`. We define the `other` sub-pattern to show extensibility of our workload-representation framework. We are able to add any further aggregation or grouping operation to our pattern framework if these are not derivable from our existing patterns. We argue that this extension is applicable for both other pattern groups as well; whereby we do not exclude a new pattern group.

In summary, we extend our workload representation for the relational algebra by a new group of sub-patterns and the redefinition of existing (sub-) patterns only. First, we add aggregation and grouping operations to the new aggregation & grouping pattern in different characteristics to support the SQL syntax. Second, we redefine and extend the tuple-operations pattern to support different architectures of SQL-minted systems. We recognize different tuple-processing schemes, make data-flow analyses possible, and consider architecture-specific operations. Third, the Cartesian product and its derivable operations are covered by two different concatenation types: (a) tuple-wise value comparison of join attributes whereas intermediate results are produced on necessity immediately and (b) vector-wise whereas intermediate-result materialization is suspended. Implementation details, as different join schemes (e.g., merge join), are not covered here. Due to the support of relational algebra operators and SQL syntax, we conclude that our workload-representation approach allows us to map each workload operation to our workload patterns. Moreover, we are able to introduce other new relational architectures into our framework and compare them to the existing Column Stores and Row Store approaches.

4.2.3. Administration, Analysis, and Aggregation

We establish our workload-representation framework to administer workload statistics for relational DBMSs above. We are aware, each DBMS administers its own statistics in some manner. However, we aim at minimally-invasive approach as well as at architecture-independent statistic administration. We show the architecture independence of our approach in Section 4.3.

We argue that a standalone approach fosters both architecture independence and minimal overhead in existing systems. That is, we achieve both objectives with a standalone approach. Whenever we administer statistics of one (the system itself) or more systems in an existing system; we cause additional workload on this system. We preserve systems that will be evaluated from additional workload with our standalone approach. Moreover, statistics from different systems are not usable from respectively in one statistic-storage system (e.g., incompatible policies). In a standalone statistic-storage system, we are able to homogenize different statistics to a representation that is sufficient and applicable to each DBMS (cf. Section 4.4). Our approach also allows us to store sample statistics. That is, we also use estimated statistics (with uncertainty) in absence of the corresponding DBSs. We state that we are able to compare physical present DBSs with samples (non-physical DBSs) as well as compare samples to each other. In summary, we achieve a maximum degree of freedom for statistic analyses and their comparability (cf. Section 5.1 for more details).

We store workload statistics or samples for different DBMSs in our framework in the same information content as DBMSs themselves. Consequently, we are able to analyze statistics with same methods; and alike, we are able to use same algorithms. We process query optimization like DBMS' query optimizer and additionally optimize for several DBMSs architecture-independent and in parallel. We further argue that complex analyses (e.g., statistical- and machine-learning algorithms [HTF09])

on stored statistics are feasible in our standalone solution, but in a DBMS itself such analyses cause too much impact on current workload. However, we enable new investigation methods (e.g., bottleneck detection, load balancing) based on the presented statistic-representation approach. Therefore, we need analyses across different operations, queries, resources, and architectures.

We further argue, DBMSs support different degrees of detail concerning statistics; whereby statistic samples mostly do not achieve a comparable detail degree as existing DBSs. Consequently, we have to process given statistics to support their comparability whenever we observe different degrees of detail. We focus on bottom-up alignment of different detail degrees due to the fact that the vice-versa approach (top-down) has to compute artificial values to achieve more detailed statistics. Artificial statistic values increase the uncertainty for estimations and reduce the confidence of our approach. In consequence, we transform all statistics of the DBMS that supports more detailed statistics, to the lower degree of detail of the DBMS to be compared. Nevertheless, we are able to satisfy each (feasible) degree of detail for statistics. Our freedom of choice spans from a single operation of a query (most fine-grained detail) to the three coarse workload patterns (i.e., join, tuple-operation, and aggregation & grouping) for statistics to be compared. Beside single operations of a query, we support various statistic-aggregation levels (bottom-up only) in our framework to compare DBMSs (i.e., their cost estimates) query-wise, sub-pattern-wise, and each permutation of these levels up to very abstract costs for complete workloads (e.g., three coarse workload patterns). That is, the analysis methods and their degrees of freedom are only limited by the detail degree of given statistics. We give further insights in Section 4.3. We conclude that our approach supports each more or less complex analysis with respect to the given degree of detail for stored statistics.

4.2.4. Threats to Validity

We introduce our workload decomposition and representation framework based on our own abstraction level and our own point of view. That is, we do not claim completeness. We are aware that we do not observe each relational algebra and/or SQL extension above. We argue that we consider the most important processing schemes (i.e., join processing, column- and tuple-wise processing, and aggregations) and we reduce more complex operations to our workload patterns. That is, we reduce the complexity of workload analyses and prune the solution space (number of operators and operations), in line with others [KBL05, Pages 128 ff.] and to the best of our knowledge, to a sufficient solution. Nevertheless, we observe dependencies between several patterns. We consider the join, the filtering, the sort/order, the group by, the cube pattern, as well as the tuple reconstruction, and the data access in the following.

First, we consider the complexity of join operations. Join operations inherently imply tuple selections (e.g., equality of join-attribute values). Hence, we may map these selections to the filtering pattern. However, the tuple selection itself is part of the join operation [KBL05, Pages 137 ff.] by definition [Cod70]. Moreover, we need

to implement new techniques to further decompose join operations and gather the necessary statistics for concatenation of tuples and the selection due to the fact that SQL-minted systems do not reduce joins to the Cartesian product and selection of tuples[8]. Hence, the administrative cost for tuning would be noticeably increased. We state that an additional decomposition of join operations is not necessary. Consequently, we keep tuple-selection-cost mapping of joins to the join pattern. To a side-effect, we sustain the comparability of join techniques according to different architectures because of system-specific decomposition of join operations is not necessary.

Second, we state that two different types of sort/order operation can occur in workloads (i.e., implicit and explicit sort). We distinguish between explicit and implicit sort because the explicit is evoked by workload or a user (command) and the implicit sort is evoked by the optimizer (reduce query costs). We map the explicit sort operation to the sort/order pattern due to the fact that each DBMS have to process this sort for queries in any way. In contrast, the implicit sort (e.g., for sort-merge join) does not have to be executed in each DBMS because each optimizer follows different query-optimization policies (e.g., sort-merge join vs. hash join). We argue that join operations do not only evoke implicit sorts but also groupings, cube operations, and aggregations may evoke sort operation. In summary, we assign the costs for implicit sorts to the corresponding (parent) operation (e.g., GROUP BY) to sustain comparability between different architectures/DBMSs.

Third, we determine that the tuple reconstruction is a column-store-specific operation and no optimization policy. Column Stores have to reconstruct tuples for several operations (e.g., multi-column selections, sub-queries) whenever a query computation refers to more than one column (cf. Section 2.2 and/or Chapter 3). In contrast to optimization policies (e.g., sorts), the reconstruction of tuples is not optional but mandatory with respect to the architecture. We emphasize that tuples are found in Row Stores inherently and Column Stores have to reconstruct them at a certain point during query processing. That is, we distinguish between tuple reconstruction and operations which cause the reconstruction to represent the architectural differences of Column Stores and Row Stores. Consequently, we sustain the comparability of operations beyond the architectures. We assign the tuple-reconstruction costs to the data-access sub-pattern as we discussed above due to the fact that these costs correlate to the saved I/O on data access.

Summarizing, we are aware of dependencies between database operations respectively our patterns and discuss the mapping of these in our approach. We argue that our approach is suitable and further extensible for relational DBMSs. Nevertheless, we do not claim completeness, but we show our approach's architecture independence in the following sections.

[8]The DBMS optimizer selects the most efficient join implementation to avoid Cartesian products during query execution.

4.3. Architecture-independent Workload Representation

We present our workload-representation framework concerning Row Stores (i.e., Oracle) and the relational algebra in the previous section. However, we argue that we do not need a separate decomposition algorithm for Column Stores (i.e., our workload patterns are also sufficient to query plan operations of Column Stores) because operation naming in Column Stores only differ from typical naming in Row Stores but the abstract processing schemes are equal (cf. Sections 2.2, 4.1, and 4.2). We show the architecture independence of our approach by two methods. First, we representatively illustrate the mapping of query-plan operations for a well-known Column Store from research (C-Store/Vertica[9] in our case). Second, we practically illustrate our approach by representation of column- and row-store statistics (in our case – Infobright ICE[10] respectively Oracle) in our workload-representation framework.

First, we present the query-plan operations introduced in [SAB+05] and the mapping of operations to our workload patterns as follows:

Decompress: The decompress operation decompresses data for subsequent operations in the query plan that cannot be processed on compressed data (cf. [Aba08]). That is, we map this operation to our data-access pattern.

Select: The select is equivalent to the selection in the relational algebra except that selection's result is represented as a bit string. Hence, we assign the select to our filtering pattern.

Mask: The mask operation is defined on bit strings and only returns those values whose corresponding bits in the bit string are (represented as) 1. Note, C-Store uses different encoding schemes for columns [SAB+05]. One of these encoding schemes is the bit representation in bitmaps (cf. Sections 2.4 and Chapter 3). We argue that this behavior corresponds to a specialized selection. Consequently, we map the mask operation to our filtering pattern.

Project: The project is equivalent to the projection of relational algebra. Thus, we assign this operation to our projection pattern.

Sort: This operation sorts columns of a C-Store projection according to a (set of) sort column(s). This operation is equivalent to sort operations on projected tuples. That is, we map the sort to the sort/order pattern.

Aggregations: These operations compute aggregations (e.g., SUM) and groupings (e.g., GROUP BY) equivalent to SQL [Aba08], thus, we directly map these operations to the corresponding sub-pattern in our aggregation & grouping pattern. Note, Column Stores (in our case C-Store) compute only necessary

[9] Vertica is based on research on C-Store.
[10] `https://infobright.com/`.

column for aggregations in contrast to Row Stores that compute complete tuples.

Concat: The concat operations combines C-Store projections that are sorted in the same sequence into a new projection (cf. Sections 2.4 and Chapter 3). We consider this concatenation of (sets of) column(s) as specific type of tuple reconstruction operation and map it to the corresponding (data-access) pattern.

Permute: This operation permutes the order of columns in C-Store projections according to the given order by a join index (cf. Section 2.4). It prevents additional replication overhead that emerges through creation of join indexes and C-Store projections in several orders. This operation is used for joins, thus, we map its cost to our join pattern. Furthermore, the join type determines to which sub-pattern we assign these costs (see below).

Join: We map this operation to the join pattern and distinguish two join types. First, if tuples are already reconstructed then the join is processed like in Row Stores. That is, we map this join type to the non-vector-based sub-pattern. Second, the join operation only processes on columns that are necessary to evaluate the join predicate. The join result is a set of position pairs of the input columns where the predicate applies [Aba08]. The second join type may process on compressed data as well as it may uses vector-based join techniques [SAB+05, Lüb08]. That is, we map this join type to the vector-based sub-pattern.

Bitstring operations: Bitstring operations are logical operators (AND, OR, NOT) and are only defined for bit strings. These operations process two bit strings and compute a new bit string with respect to the corresponding logical operator (i.e., bitwise AND respectively OR, complement). In other words, the bitstring operations implement the concatenation of different selection predicates. Hence, we map these operations to our filtering pattern.

We show that our approach is also applicable for a well-known column-store implementation from research. We illustrate the mapping of the above discussed operations to our approach in Figure 4.11. We argue that our scenario shows according to the architecture independence soundness for our approach. Nevertheless, we present our second pass for architecture independent statistic representation in the following.

Second, we show the soundness of our approach with a test environment. Therefore, we use our running example DBMSs. That is, we use Oracle (i.e., 11gR2 Enterprise Edition) as row-store representative and use Infobright ICE 3.3.1[11] as column-store representative. We use the TPC-H benchmark [Tra10] (i.e., version 2.11.0 with scale factor 1) for our test environment. We again decide for the read-only (OLAP) benchmark TPC-H due to the fact that Infobright ICE is a read-only DWH [Inf08, Inf11a]. We perform all 22 TPC-H queries on both systems, gather the statistics, and

[11] Note, ICE 3.2.2 was no longer available for download. Nevertheless, our results are comparable for ICE, thus, we do not redo previous experiments.

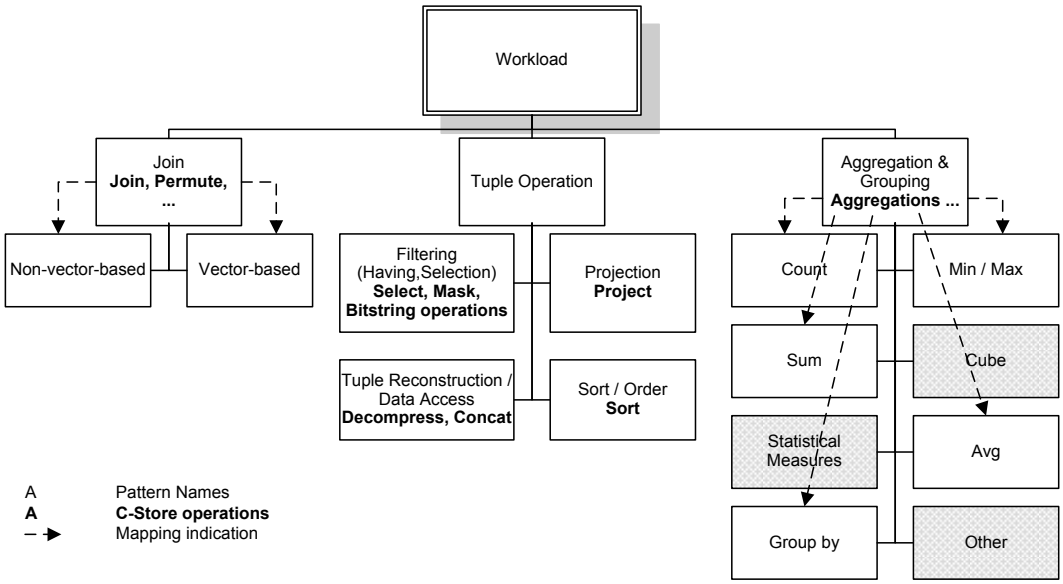


Figure 4.11.: *Workload patterns with C-Store-operation mapping (inspired by [SAB+05, Aba08]).*

store the statistics in our framework. We go into detail in the following paragraphs. For reasons of clarity and comprehensibility, we show only three representative TPC-H queries[12] namely `Q2, Q6,` and `Q14` in our test scenario (cf. Figure 4.12). We present the results for the remaining queries in Appendix A.2[13].

The query structure, syntax, and execution time are not sufficient to estimate the query-performance behavior on different storage architectures (cf. Chapter 3 and Chaudhuri et al. [CKR05]). We introduce an approach based on database operations that provides analyses to find long running operations (bottlenecks). Moreover, we want to figure out reasons for bad (or good) performance of operations in DBMSs, thus, we have to use additional metrics. We select the I/O cost[14] to compare DBMSs and summarize the optimizer outputs in Table 4.2. We state that I/O cost is a reasonable cost metric but is not sufficient to select the optimal storage architecture. We will show this effect for I/O cost with a negation example in the following. Following our previous name convention, we define the query `IDs` according to their TPC-H query number (i.e., we map the queries with the `IDs 2`, `6`, and `14`). The operations are identified by their query plan number (IDs in Table 4.2). Thus, the root operation of TPC-H query `Q2` has the `ID 2.0` in Figure 4.12. All values in Table 4.2 are given in KBytes. The given values are input cost of each operation except the table access cost because no information on input cost to table access operations are available. Note, the granularity of Oracle's cost measurements is

[12]The queries show typical results for the TPC-H benchmark in our test environment.
[13]Please cf. [Tra10] for the complete schema and query description.
[14]I/O cost is a best practice cost metric.

on the byte level whereas the measurements of ICE are on the data pack (65K) level [Inf11b]. Nevertheless, we used the default data block size 8KBytes in our Oracle installation; that is the smallest accessible unit.

In Figure 4.12, we present the workload patterns with I/O cost[15] – the optimizer output – of the corresponding TPC-H queries. As mentioned before, the projection operation causes no additional cost. Hence, the I/O cost in Table 4.2 and Figure 4.12 represent the size of final results. The stored information can be analyzed and aggregated in decision models with any necessary granularity. In our example, we only sum up all values of the data-access pattern for each query to compute I/O cost per query in KBytes. For the three selected queries, all results and intermediate results are smaller than the available main memory, thus, no data has to be reread subsequently. That is, we refer to data to be read from disk for the following consideration. We suppose, the DBMS with minimal I/O cost performs best (as we mentioned before, I/O cost is a good cost metric). Oracle reads 1,452.133 KBytes for query `Q2` and takes 8.14 seconds. ICE needs 41 seconds and accesses 2,340 KBytes. The results for `Q2` support our assumption. Nevertheless, we cannot confirm our assumption for query `Q14`. Oracle accesses 7,020.894 KBytes and computes the query in 22.55 seconds whereas ICE computes it in 3 seconds and reads 38,544.757 KBytes. Moreover, the contradiction confirms for query `Q6`. Oracle (3,118 KBytes) accesses less data than ICE (17,529.664 KBytes), but ICE (2 seconds) computes this query ten times faster than Oracle (22,64 seconds). Hence, we cannot figure out a definite correlation for our sample workload.

We have previously shown that I/O cost alone is not a sufficient metric to estimate the behavior of database operations and further, we suggest that each single cost metric is not sufficient. However, I/O cost is one important metric to describe performance behavior on different storage architectures because one of the crucial achievements of Column Stores is the reduction of data size (i.e., I/O cost) by aggressive compression. The I/O cost also gives an insight into necessary main memory for database operations or if operations have to access the secondary memory. Hence, we can estimate that database operations are completely computed in main memory or data has to be (re-)read stepwise[16]. We assume that sets of cost metrics are needed to sufficiently evaluate the behavior of database operations. Therefore, one needs tool support as we propose in this thesis.

Each relational DBMS is referable to the relational data model, so these DBMSs are based on the relational algebra in some manner, too. Thus, we can reduce or map those operations to our workload patterns; in worst case, we have to add an architecture-specific operation (e.g., tuple reconstruction for Column Stores) for hybrid DBMSs to our pattern. For a future (relational) hybrid storage architecture, such an operation could be necessary to map the cost for conversions between row- and column-oriented structures and vice versa.

[15] We are aware, the data size has to be greater than or equal to the page size.
[16] We remind of the performance gap (circa 10^5) between main memory and HDDs.

Workload	**Q2**		**Q6**		**Q14**	
	Oracle (8.14sec)	**ICE** (41sec)	**Oracle** (22.64sec)	**ICE** (2sec)	**Oracle** (22.55sec)	**ICE** (3sec)
Data Access	*ID16*:1,440 *ID15*:0.104 *ID13*:11.2 *ID12*:0.029 *ID7*:0.8	*ID24-26*:3*102.513 *ID22*:102.513 *ID21*:102.513 *ID19*:1,332.664 *ID18*:410,051 *ID16*:1,332.664 *ID15*:410.051 *ID13*:410.051 *ID12*:410.051 *ID6-11*:6*102.513 *ID5*:410.051	*ID2*:3,118	*ID5*:4,382.416 *ID4*:4,382.416 *ID3*:4,382.416 *ID2*:4,382.416	*ID4*:5,400 *ID3*:1,620.894	*ID8*:410.051 *ID7*:9,533.676 *ID6*:9,533.676 *ID5*:9,533.676 *ID4*:9,533.676
Non-vector-based	*ID11*:11.229 *ID10*:17 *ID9*:88.016 *ID8*:1,440 *ID6*:202.760	*ID23*:205.025 *ID20*:205.025 *ID17*:1,742.715 *ID14*:1,742.715			*ID2*:7,020.894	*ID3*:9,841.214
Tuple reconstruction		*ID4*:512.563 *ID2*:717.588				*ID2*:410.051
Sort	*ID5*:45.346 *ID3*:33.18	*ID3*:410.051 *ID1*:820.101				
Count	*ID1*:31.284					
Sum			*ID1*:3,118	*ID1*:4,382.416	*ID1*:3,610.173	*ID1*:205.025
Projection	*ID4*:45.346 *ID2*:33.18 *ID0*:19.8	*ID0*:820.101	*ID0*:0.02	*ID0*:102.513	*ID0*:0.049	*ID0*:102.513

Table 4.2.: *Accessed KBytes by query operations of TPC-H query Q2, Q6, and Q14.*

Figure 4.12.: *Workload graph with mapped I/O cost of TPC-H query Q2, Q6, and Q14.*

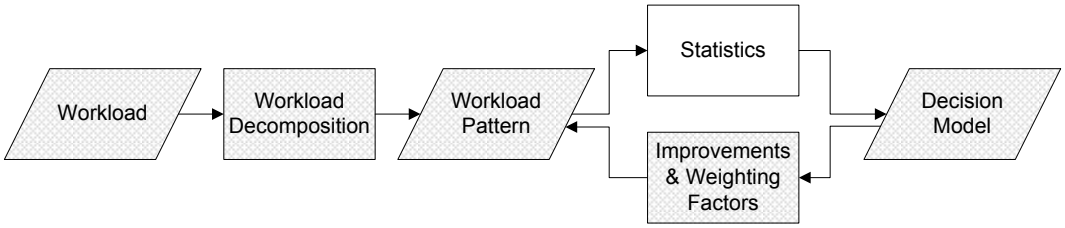


Figure 4.13.: *Workflow – Chronological classification for the statistic normalization policies.*

4.4. Statistic Normalization

In previous discussions, we do not consider different cost representations that we extract from query plans (i.e., different DBMSs/optimizers). That is, we show workload statistics independent from storage architecture and DBMS in our approach. We state that the extracted statistics have to be transferred into a homogeneous presentation, thus, we ensure sound comparison of cost values (cf. Figure 4.13 for framework classification). We present our policies to normalize different cost representations in the following.

We state that it is necessary or at least advantageous to use measurable values for cost estimation. Artificial computed costs are often inexplicable due to the fact that the computing method is mostly undisclosed (e.g., the Cost value in Oracle (cf. Section 4.1)). That is, we cannot reproduce these cost computations. Whenever we use artificial computed costs, we can only compare equal DBMSs or at least DBMSs from the same vendor. Moreover, we cannot compare performance of either different DBMSs or different storage architectures. We argue that cost measures are not only applicable for performance estimation, but for query optimization and/or algorithm selection (e.g., cardinality). As we state before, our approach is transparent to optimization policies. That is, we focus on resource-consumption measures. In the following, we show exemplary normalization for ***I/O*** and ***CPU*** cost.

First, we present our ideas to I/O-cost normalization (cf. also [Lüb09]). We state that I/O is specified in a storage-space-consumption unit (e.g., Oracle specifies I/O in bytes). The standard storage-space-consumption units are multiples of bytes, thus, we can easily compute the values to a uniform unit (cf. Equation 4.1).[17] We argue that we automatically achieve compatibility between measured and estimated values in consideration of approximation errors with our approach due to the fact that either measured or estimated I/O is represented as multiple of bytes.

$$\begin{aligned} \text{1,099,511,627,776 Byte} &= \text{1,073,741,824 KByte} = \\ \text{1,048,576 MByte} &= \text{1,024 GByte} = \ldots \end{aligned} \tag{4.1}$$

However, we determine that some DBMSs do not use an I/O representation in

[17]Note, the conversion factor for storage-consumption units can be 1,024 or 1,000. We use 1,024.

$$\#\text{ storage units} * \text{storage-unit size} = \#\text{ KBytes}$$

$$\text{(a) } \#\text{ storage units} * 8\text{ KByte} = \#\text{ KBytes}$$

$$\text{(b) } \#\text{ storage units} * 32\text{ KByte} = \#\text{ KBytes}$$

$$\text{(b)}^{\star} \#\text{ storage units} * \text{compressed KByte} = \#\text{ KBytes}$$

$$\text{with } \frac{65{,}536 * \textit{data-type size}}{1{,}024 * C/R} = \text{compressed}$$

Figure 4.14.: *Conversion for I/O normalization based on DBMS-storage units.*

$$\text{Numbers: } \lceil \#\text{Bytes} \rceil = \frac{\textit{length}\,(p) + s}{2} + 1 \text{ with } p = \text{precision},\ s = \text{sign flag } (0/1)$$

$$\text{Characters: } \lceil \#\text{Bytes} \rceil = \textit{length}\,(\textit{size}) \text{ with (n)char}(\textit{size}),\text{(n)varchar2}(\textit{size}), \ldots$$

Figure 4.15.: *Compute (approx.) byte size of data types in Oracle [Tan08, Ora10b].*

terms of byte multiples. These DBMSs use the number of accessed (minimal) DBMS-storage units (e.g., pages/blocks) to represent the I/O.[18] In consequence, we have to know the smallest accessible DBMS-storage unit. We observe two implementation groups concerning storage units in DBMSs: (a) the DBMS uses a parameter (e.g., Oracle) to set the size of the smallest storage unit (i.e., page/block size) or (b) the DBMS uses a fixed sized storage unit (e.g., LucidDB or Infobright ICE). Nevertheless, we obtain the same computation method for (a) and (b). In Figure 4.14, we show the general computation method and in each case one example. We use for (a) a standard parameter for many DBMSs (8KByte, e.g., Oracle, PostgreSQL) and for (b) fixed page size for LucidDB[19] (32KByte). We argue that the storage-unit size is available from DBMS configuration or at least from documentation in terms of fixed storage units, thus, we can also normalize these I/O values. However, we detect specific DBMSs, like Infobright ICE [Inf08, Inf11a], that put together a certain number (e.g., 65,536 values for Infobright) of column values independent from their storage size (i.e., this storage behavior is specific for Column Stores).

We have to observe the storage size of each column value to compute the storage size of joined column values. We get the approximated column-value size by data type of the corresponding column because we can derive the byte-storage size from the data type. That is, character-data types require (approximately) 1 byte per sign on the one hand. On the other hand, number-representations use fixed byte-size from catalog (e.g., an integer value equals 4 byte) or the size is dependently computed by numbers length (cf. [Syb10, Pages 69 ff.] for Sybase). We argue that these two representations are sufficient because (almost each) more complex data type can be reduced to simple char or number representations (e.g., as in our case

[18] Some DBMSs also use both I/O representations (e.g., Oracle).
[19] `http://www.hydromatic.net/wiki/LucidDbSystemParameters`.

in Oracle [Ora10b]). We show examples for Oracle in Figure 4.15 (cf. [Tan08] for details).[20] However, the joined column values are also compressed, thus, we achieve a Compression Ratio (C/R). The C/R is variable in our computation method, thus, we may use this method for other DBMSs, too. We figure out C/R-values from 10:1 to 40:1 in Infobright's information material [Inf08, Inf11a]. Concerning our experiences (cf. Section 2.2 and Chapter 3), a C/R 10:1 is most applicable as approximation. However, the C/R is adaptable for different data sets and/or DBMSs easily. We present the adopted computation method for Infobright in Figure 4.14 (b)*.

In conclusion, we have to use both I/O-normalization approaches whenever DBMSs do not support the same (or both) I/O representation(s) ((a) and (b)). Finally, we argue that each presented method is an approximate technique in some way. The I/O specification in byte as well as number of storage units[21] does not exactly map the filling of pages. In other words, each page is not equally or completely filled even though the DBMS parameters concerning free space per page and deallocation threshold of pages are known. To reduce this uncertainty, we measure the complete database size and count the number of all allocated pages. This approach is limited to existing DBSs as well as an approximated filling of pages from the DBMS itself which some DBMSs support on its own. The same idea is applicable for dynamic allocated data types (e.g., varchar). Our approach considers the maximum length of dynamic data type that is not true in general. Therefore, we use the average length of dynamic data types to improve the soundness of our computation. The average length of data types can be computed in existing DBSs or approximated from samples.

Second, we show precise and approximate conversion methodologies for CPU-cost representations in DBMS optimizer. We state that precise cost-representations[22] are CPU cycle and CPU time [HP12]. Both cost representations are very common for DBMSs and OSs. We convert CPU cycles to CPU time and vice versa by the help of CPU clock rates[23]. We show the conversion formula in Figure 4.16. However, we abstract from CPU-hibernation techniques (and akin). On the one hand, we state that CPU startup is very quick from hibernation, and moreover CPU-startup time shall be equal for a sound evaluation due to equal DBS hardware. On the other hand, we argue that in most cases DBS's hardware do not hibernate during workload processing. We state that the unique CPU-startup is negligible at the beginning of workload processing.

Nevertheless, we observe DBMSs that use an approximate CPU-cost value the CPU utilization (e.g., Oracle). CPUs are either busy (100%) or idle (0%). Due to the fact that systems hardware scales equally[24], CPUs alternate between busy and

[20] Note, our example computation refers to Single-Byte character sets. For Multi-Byte character sets size equals to bytes [Ora10b, Ora10c].
[21] Except we may exactly measure these values.
[22] We are aware that the CPU costs are approximations from DBMS optimizer. We refer to the conversion type with the accuracy class.
[23] Please also cf. [HP12, Pages 48 ff.] or computation in MATLAB [FM02, Pages 374 f.].
[24] Remind the performance gap (approx. 10^5) between main memory and HDDs.

$$\text{CPU time} = \frac{\text{CPU cycles for an operation}}{\text{clock rate}}$$

$$\frac{\text{CPU utilization}}{\text{operation}} \approx \frac{\text{CPU time for an operation}}{\text{total time for an operation}}$$

$$\frac{\text{CPU utilization}}{\text{query}} \approx \frac{\sum \text{CPU time for operations}}{\text{total time for a query}}$$

Figure 4.16.: *Conversion of (approx.) CPU-utilization units (time, cycles,...) for different DBMSs following others [FM02, Pol08, HP12].*

idle state in sequential processing mode. In other words, tasks alternate between busy (processing) and idle state (e.g., CPU waits for I/O). Therefore, OSs and/or DBMSs schedule different tasks in parallel to avoid CPU waits (i.e., idle states). That is, the CPU utilization is the percentage that an operation or a query (i.e., a task) utilizes the CPU within a defined time slot [HP12, Pages 48 ff.].[25] We state that the CPU utilization is an approximate cost value because we are not able to convert CPU utilization values into other CPU-cost representations. We need empirical or measured values (i.e., time (slots)) to approximate other representations (e.g., CPU time) as well as vice versa. We present an approximate computation for each per operation and per query in Figure 4.16. We argue that CPU utilization is the most imprecise CPU-cost representation and implies an information loss compared to CPU time and CPU cycles. Our approach additionally implies the compatibility of optimizer costs and measured cost values. That is, (a) CPU time, CPU cycles, and CPU utilization are measurable in OSs and/or DBMSs and (b) DBMS optimizer estimates CPU costs that we convert into the corresponding (of the three) CPU-cost representations. Finally, we remark that further conversions and approximations methodologies for CPU costs are available (e.g., by help of instructions per cycle (IPC)/cycles per instruction (CPI)). Nevertheless, we do not further elaborate on this issue and refer to [HP12]. For cost estimation with uncertainty, we refer to Fortier and Michel [FM02] as well as Poladyan [Pol08].

In conclusion, we normalize measurable costs to achieve a comprehensible and comparable statistic representation for relational DBMSs in general. Furthermore, we guarantee comparability between measures and estimations with our normalization process. We do not claim completeness that we use all representative cost measures. To the best of our knowledge, CPU consumption and I/O are the most important cost measures for DBMS performance estimation, thus, we present the normalization process for these two measures. Nevertheless, our approach is transparent to the underlying costs because the normalization policy is applicable to each (measurable) cost value. We show ideas concerning uncertainty and soundness that can further

[25] A thought example can be found at `http://www.ibmsystemsmag.com/ibmi/administrator/performance/Calculating-CPU-Utilization/`.

```
1 oracle = ConnectionManager.getConnection(/*your connection*/);
2 xplanstmt = oracle.createStatement();
3 xplanstmt.execute("EXPLAIN PLAN FOR /*put in your query here*/");
4 xplan = xplanstmt.executeQuery("SELECT plan_table_output FROM table(dbms_xplan.display())");
5 while (xplan.next())
6 {
7   System.out.println(xplan.getString(1));
8 } ...
```

Figure 4.17.: *Exemplary explain plan execution via JDBC for Oracle.*

improve our approach. We state that these (existing) approaches are not in the focus of our work; the approaches may be easily implemented in our approach later due to modular and transparent implementation of the workload representation approach.

4.5. Details of Implementation

We omit implementation details in the previous sections. In this section, we show the way to easily reimplement our approach. That is, we show the repeatability of our approach without our purpose-built implementation and/or previous work (e.g., QGM briefly introduced in Section 4.2 [Bub07, LGB09]). Furthermore, we assume higher soundness of our approach due to the fact that we eliminate artifacts from implementation. We divide our approach into three parts: 1) ***statistics extraction***, 2) ***parsing and normalization***, and 3) ***storage***. In abstraction, we can classify the first two parts as ETL component and the third part as data warehouse database in a (reference) DWH scenario [Inm05]. We argue the three parts are completely transparent to each other, to predecessor systems (e.g., statistic sources), and to successor systems (e.g., analysis tools).

First, we extract the statistics from DBSs. Therefore, we observe two approach classes: (a) interface-based approaches (e.g., JDBC/ODBC [SGS03, SSG04] or R [RWM02]) and (b) integrated system tools[26] (e.g., SQL*Plus (Oracle), mysql(-ib) (MySQL/Infobright), or dbisql (Sybase)). We show an exemplary statistic extraction for Oracle (a) via JDBC in Figure 4.17 and (b) with command-line tools [GNU10] via SQL*Plus in Figures 4.18 and 4.19. However, we have to know for each DBMS where query plans and statistics are stored internally. Additionally, we have to prepare optimizer output for analysis, whereby the optimizer output is individually configurable for many DBMSs.

Second, we have to parse and normalize the extracted data to an interpretable representation. Therefore, we use Bash command-line tools [GNU10] that drop unnecessary information and normalize cost values. Finally, we prepare the simplified cost representation for operation-wise import in a database schema[27]. We provide

[26] Vendor tools (e.g., Oracle SQL Developer) are special cases of this class because they use optimized vendor-specific interfaces.

[27] The distribution of operations to tables is dependent on the given schema.

```
#!/bin/bash
echo
i=1
for i in {1..n}; do #n represents number of queries
sqlplus yourusername/yourpassword < explain$i.sql
```

Figure 4.18.: *Exemplary explain plan execution via Bash and SQL*Plus for Oracle.*

```
rem Compute explain plan
@utlxplan$i /*Configure your optimizer output – please cf. Oracle documentation*/
EXPLAIN PLAN FOR
/*put in your query here*/;
SPOOL explain_query$i.csv /*Writes output in current path*/
SELECT * FROM table(DBMS_XPLAN.DISPLAY('PLAN_TABLE$i'));
SPOOL OFF
```

Figure 4.19.: *Exemplary explain.sql for query plan extraction via SQL*Plus.*

the code in Listing 4.20 for Oracle standard output.

Third, we show a feasible storage and administration solution for statistics that does not use our QGM implementation. We state that the usage of the alternative solution does not have downsides for decision quality. We use the QGM due to experience in preliminary work, integration of code into our framework (e.g., interface implementation), and performance issues for data aggregation (e.g., joins in the relational representation). However, we present an entity-relationship schema for our workload presentation in Figure 4.21 (inspired by Chen [Che76]).

We summarize that the recreation of statistic-extraction mechanism and workload representation are easily applicable with just little effort. By usage of scripting utility (e.g., Bash script) or high level programming languages (e.g., Java), the extraction and loading methodology can be automated easily.

```
#!/bin/bash
i=1
for i in {1..n}; do #n represents number of queries
file=/yourpath/explain_query$i.csv
#kill, header, double quotes, and unnecessary (empty) columns; clean line beginning; add INSERT INTO
  statement
tail -n +2 $file | sed  -e 's/,[\"][\"]//g' \
                    -e 's/[\"][\"]//g' \
                    -e 's/^.//' \
                    -e 's/\(.*\)/INSERT INTO "TABLE" VALUES(\1);/' > /yourpath/cleaned_query$i.csv
done
```

Figure 4.20.: *Exemplary Bash script for cleanup of optimizer output in Oracle.*

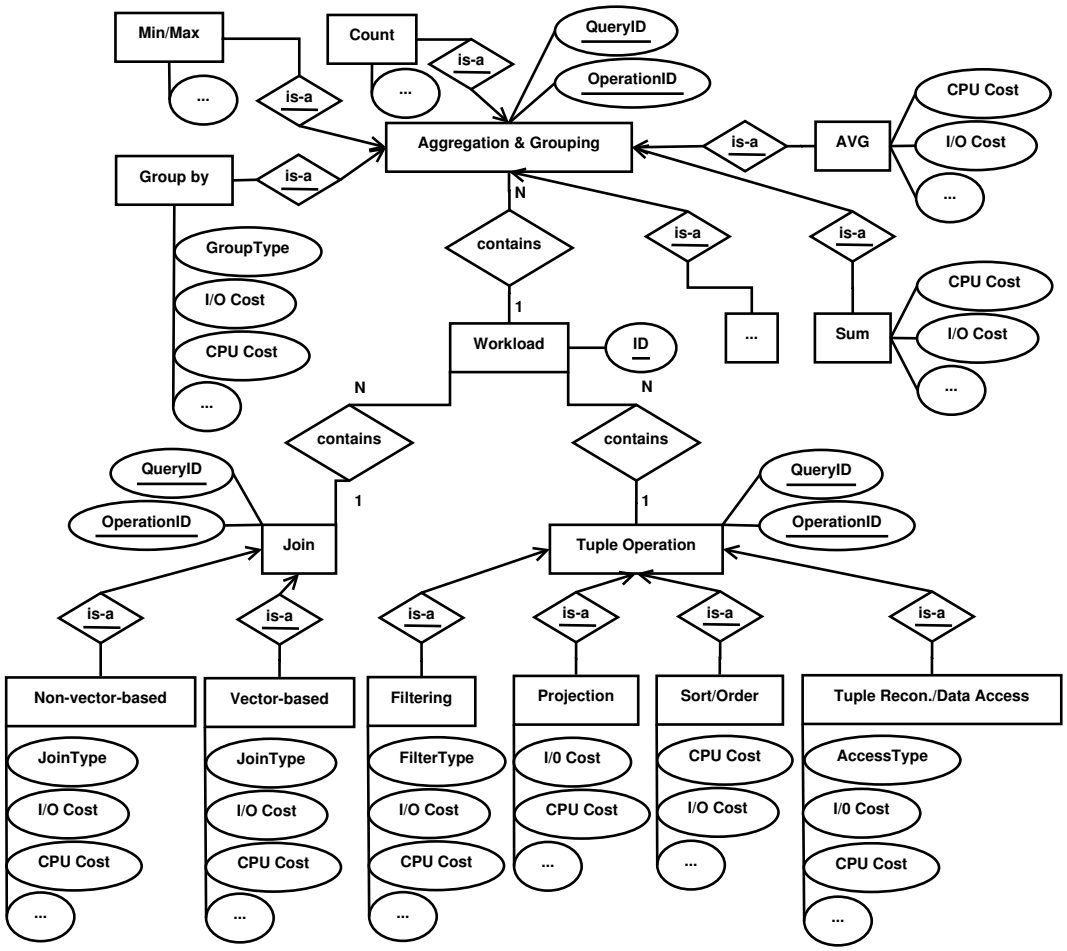


Figure 4.21.: *Entity-relationship schema for our workload-representation approach (inspired by Chen [Che76]).*

4.6. Summary

In Chapter 4, we introduced our statistic-extraction and administration approach that allows query and workload analyses for different DBMSs and storage architectures, respectively (i.e., we only observe Row Stores and Column Stores currently).

Our idea started with the relational algebra – the fundamental concept for relational DBMS; whereupon we introduced abstract base operators (cf. Section 4.1). Hence, the relational algebra enables us to create queries based on these operators in form of query plans as well as query plans are used in relational DBMSs to estimate costs of a query and figure out (or reuse) an optimized execution plan[28] for a query. In consequence, we reduced query plans from DBMS to relational algebra operators. We obtained better support of SQL in our approach due to addition of aggregations and groupings to the relational algebra operators. We used our combined operator set to represent given workloads and the corresponding execution costs that we extracted (or estimated) from DBMSs.

We administered the extracted workload statistics in our workload patterns. So, we were able to analyze and/or aggregate the statistics independently from DBMS and query execution. Therefore, we introduced patterns that represent types of operations. Different types of operations characterize different processing schemes on the relational data model (cf. Section 4.2). The processing schemes respectively workload patterns additionally depict the architectural strengths and weaknesses of DBMSs. We showed strengths and weaknesses independently from the corresponding DBMS. We derived the independence from DBMSs and their architecture from the fact that our approach is based on the relational algebra and query plans on relational (stored) data. Therefore, we followed two different ideas (cf. Section 4.3). First, we showed the mapping to our workload pattern by column-store operations from C-Store. Hence, we discussed how we mapped C-Store operations to processing schemes in terms of the relational algebra and to our workload pattern, accordingly. Second, we showed the integration of statistics from a Row Store and a Column Store by example (i.e., Oracle 11gR2 and Infobright ICE 3.3.1). We argued that a number of cost components had to be extracted and considered for convenient estimation of query execution. Nevertheless, we only showed one cost component from statistic extraction to hide unnecessary complexity and maintain the clear arrangement in our figures. We discussed the I/O cost for both DBMSs concerning three TPC-H benchmark queries and presented the I/O cost of both DBMSs in one exemplary workload graph.

In order to achieve a homogeneous and sound statistic representation for diverse DBMSs, we introduced a statistic-normalization component in our framework (cf. Section 4.4). That is, we introduced policies for measurable cost values to compute comparable cost estimations. We discussed policies for conversation of I/O and CPU cost. We summarized plain intuitive storage-size conversions in the beginning. Moreover, we introduced one policy for storage units and for CPU utilization (i.e., CPU

[28] Remind, the execution plan of queries does not have to be optimal (cf. Section 2.4).

time to utilization per query) each. We forbeared to use artificial computed costs due to the fact that we are not able to reproduce the cost computation for different DBMSs. We are aware that more cost values exist. Nevertheless, we also used uniform or at least uniform defined cost measures (e.g., number of rows, cardinality, etc.).

In Section 4.5, we showed implementation details that are not limited to our tools. That is, we showed implementation-independent and transparent concepts of the three major framework components. First, we showed the statistic extraction with two approaches. On the one hand, we used interfaces to extract the statistics from DBMSs, and on the other hand, we used integrated system tools for statistic extraction. However, one may use the extracted statistics in any fashion and/or system. Second, we normalized the extracted statistics to an interpretable and homogeneous representation for succeeding computations. That is, we could implement the parsing and normalization component in a programming language which masters string extraction and manipulation. We presented our code examples in simple UNIX-based shell script code. We assumed that an implementation may be even easier with markup- and tag-enabled programming languages. Third, we presented an alternative storage and administration approach for relational DBMSs. On the one hand, we encouraged our approach's reproducibility, and on the other hand, the alternative approach could be easier integrated into existing meta-data storage approaches. Therefore, we illustrated the semantic representation of our QGM in an entity-relationship schema. That is, one may transpose our entity-relationship schema in any relational representation with respect to DBMS specifics.

5. Cost Estimation & Storage Advisor

Chapter 5 shares material with [Lüb10, LKS11a, LKS11b, LSKS12, LKS12].

We analyze statistics to select or recommend the optimal storage architecture. The statistics are given by a sample workload or are extracted from DBMSs as we discussed in Chapter 4. In this chapter, we introduce the Decision Model (DM) based on our workload-statistic framework (cf. Chapter 4). Therefore, we discuss the impact of cost functions[1], different granularity levels, and constraints. Based on our classification, we present our DMs to select the optimal storage architecture. We introduce a basic DM, wherefrom we derive three DMs for different application scenarios – the online DM, the design-prediction model, and the benchmarking model. First, we present the online DM which only uses extracted statistics from DBMSs. Second, we present our (offline) design-prediction model to cope with predicted workloads (or with sample workloads). Third, we show the combination of both variants – the offline benchmarking model; that is, a variant that benchmarks an existing and a theoretical system. We evaluate the DM with two DBMSs and a well-known benchmark to show the feasibility of our approach. Furthermore, we show heuristic samples that we generate from computations with our DM as well as preliminary respectively evolutionary work. In Figure 5.1, we show the chronological classification of the DM in the decision-making process.

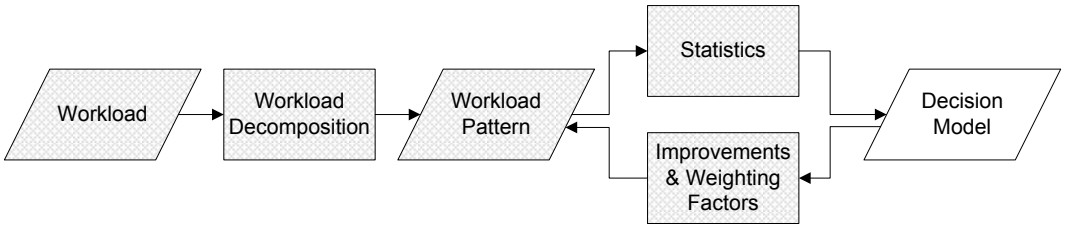


Figure 5.1.: *Chronological classification for our "advisor module" in the overall decision workflow.*

5.1. Cost Estimation with and without Uncertainty

In database tuning, we perform optimization according to a certain criterion (e.g., minimal response time for each query, maximal throughput, or average query response time for a given workload) which is most commonly done for a specific system

[1] Discussion to data source – artificial costs vs. statistic extraction – are found in Sections 4.2 and 4.4.

or architecture. Nevertheless, optimization aspects could be more complex (e.g., optimal load balancing systems, user preferences, good enough optimization to a given time threshold). A common approach is the so-called what-if analysis that we use for query optimization or physical design (e.g., index configuration [CN98, BFB14], materialized views [ZZL+04]). However, we outline an approach that is applicable across systems and architectures, for different cost functions (with and without threshold), and for samples (with uncertainty).

An optimal storage architecture has to satisfy all optimization criteria or a weighted optimum of them (e.g., majority voting). Another challenge is that there is not a single architecture outperforming the other for each database operation. Therefore, the overall optimum of all database tasks (user-specified or related to optimization criteria) has to be taken into account. We show that cost functions for single measure cannot cover all design criteria, thus, several measures (and their cost functions) have to be combined. As a result, cost functions become more complex. However, we argue that it is not useful to include all criteria into storage-architecture selection for sound results (cf. Section 4.4). Furthermore, we state that it is harder to find a sufficient cost function for more complex problems. Advantages and disadvantages of architectures may be hidden by composite cost functions (for several cost measures) due to different processing schemes (cf. Section 4.3). Moreover, the computational costs for complex cost functions can exceed the benefit for further application fields (e.g., decisions for single queries). Despite the challenges, a weighted approach for several criteria is possible; and therefore, we state that a methodology of ranking system alternatives has to be considered. An important point for ranking alternatives is the evaluation function according to given criteria. We assume, cost functions C and the corresponding queries are already partitioned in such a way that all cost values c describe the cost structure sufficiently. The costs C have to be selected according the optimization criterion (e.g., query-execution or CPU time, data size, or user preferences; cf. Chapter 4).

For the selection of the optimal storage architecture, we assume that optimization criteria are represented by cost measures (e.g., I/O) without uncertainty. Therefore, the systems to be compared have to be available. The cost function C maps a cost measure with respect to a given cost criterion. That is, we compute each cost measure with at least one cost function. Independent from the number of cost measures, we may have different cost functions for the same cost measure (cf. Figure 5.2) due to the fact that, a cost measure can be determined in continuous units as well as discrete units. Whereas linearity in a cost function (e.g., I/O in Byte; cf. Figure 5.2 – top left) results from algorithms with linear complexity which is analogously true for square root and quadratic functions. A staircase or stepwise linear function represents query-execution times where involved data has to be loaded and reallocated (e.g., disk swapping or allocate of new space; Figure 5.2 – top center and top right). Mostly, we assume that mixed or composite cost functions are common in practice. Moreover, a cost function may be discrete due to the fact that the corresponding cost measure is measured in discrete units (e.g., page or data-block size).

For small sized queries a quite different cost function behavior is appropriate than

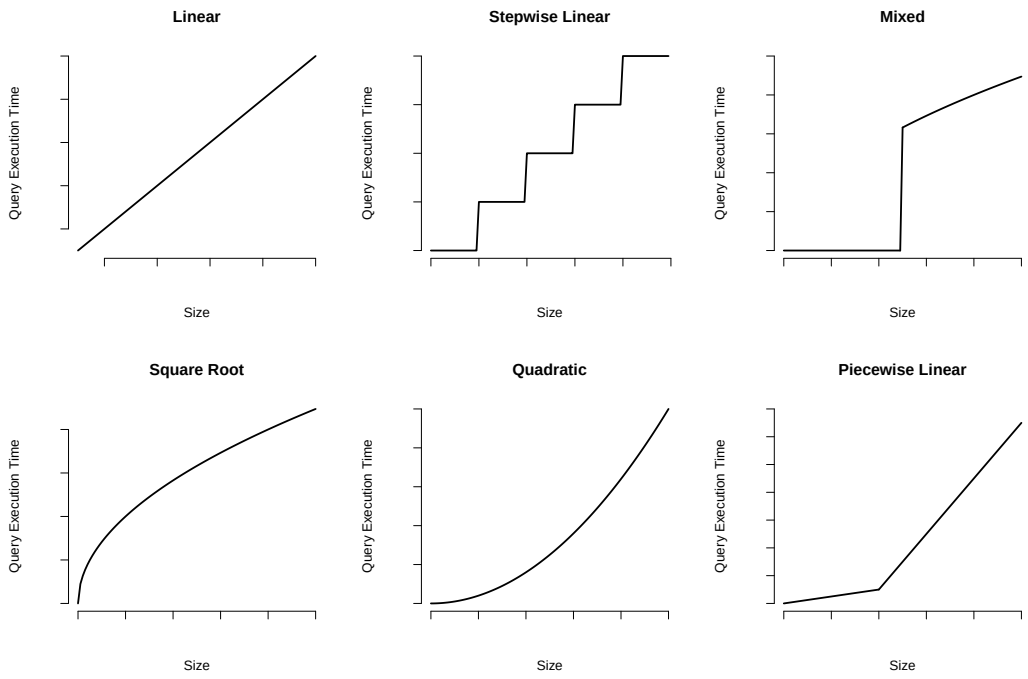


Figure 5.2.: *Different cost functions for the data size and the corresponding query execution time.*

for larger queries with respect to limited (exclusive) resources (e.g., main memory). We assume that evaluation criteria are defined. For further discussion, we assume that evaluation criteria are already defined, thus, we do not pay further attention on these. For ease of explanation and due to the fact that CPU and I/O cost are most suitable cost measures for architecture-independent comparison – what we already discuss in Section 4.4, we only use CPU and I/O cost for the remaining thesis. Nevertheless, a cost function can be used that represents user-specified measures (as well as standard cost measures) for the comparison of storage architectures. In Figure 5.2, we depict a selection of cost-function structures concerning query execution time without a claim to generality. However, we argue that our approach is transparent concerning cost functions, thus, we compute the storage-architecture selection with arbitrary cost functions (and measures) on arbitrary degree of detail. We obtain the arbitrary degree of detail due to arbitrary granularity of workload representation (cf. Section 4.2).

We have to consider the degree of detail for storage-architecture selection with uncertainty, too. We use estimated workload statistics (i.e., samples or predicted workload) whenever to be compared systems are not available. Nevertheless, we map samples to our workload-representation as we map extracted optimizer statistic approach on arbitrary degree of detail (cf. Section 4.2). We argue, the uncertainty is represented in our samples, thus, we do not directly map the uncertainty to the

cost function C. That is, our sample consists of a set of tasks which are represented by C and a probability value. The probability value describes the probability that the corresponding task is executed and encapsulates samples uncertainty. We argue that we represent samples with uncertainty sufficiently with this approach. However, we support analyses on arbitrary degree of detail (e.g., aggregate single operations to pattern level), thus, we have to compute uncertainty from the sample as well. Therefore, we suggest an error-propagation approach from uncertainty analysis (cf. Reddy [Red11] as exemplary application in energy engineering or standard literature by Meyer [Mey92]).

In our design-decision approach, we distinguish between online and offline decisions. We assume, the information space is without uncertainty for online decisions (i.e., the database system, its optimizer estimates, statistics, etc. are available). We develop for this scenario our *online DM* (cf. Section 5.2.1). For offline decisions, uncertainty has to be taken into account. We state that uncertainty results from unknown workload (parts) and/or estimation of the corresponding cost. Therefore, we develop *two offline DMs* that first, support the design or redraft of systems (cf. Section 5.2.2), and second the benchmarking of different systems (cf. Section 5.2.3). Due to the distinction between (both) architectures, we consider decision problems as ranking of architectural designs. Challenges of rankings under uncertainty are addressed, e.g., in [BK10], which we do not discuss in this thesis in more detail.

5.2. Storage Advisor: A Priori Storage-Architecture Selection

We considered gathering and storage of data (i.e., workload statistics) from different sources (cf. Chapter 4). In this chapter, we establish our storage advisor by means of DMs. Therefore, we integrate our workload patterns and therein present statistics[2] in our DM. Our DM allows us to recommend the optimal storage architecture. However, we will present basic definitions first. We will derive therefrom three DMs at abstract level and describe these in abstract, transparent, and implementation-independent way (e.g., cost functions can be replaced modularly). We remind, our workload-pattern approach is adaptable on (sub-) pattern as well, which can be added or refined (cf. Sections 4.2 and 4.4). Hence, our DMs can be further refined (e.g., more fine-grained or more coarse-grained).

We state that for physical design a common approach for enumeration[3] is the Knapsack [ACK+04] (cf. [Bru11, 92 ff.] for detailed discussion). We refer to bottom-up strategies only due to the fact that a global optimal configuration is not known beforehand. A global optimal configuration is necessary for so-called top-down approaches. However, Knapsack problems [KPP04] are specific problems of the combinatorial optimization (for details cf. [Sch03]). Enumeration by Knapsack is well

[2] Extracted from the optimizer or estimated workload statistics (samples).
[3] That is, we select a set of candidates according to given constraints.

known for decades (e.g., [Cab70]). We present the (0-1) Knapsack problem in Equation 5.1 [Pis95] that describes a maximization problem for profit p of candidates' j. x_j is binary and describes whether candidate j should be taken or not. Furthermore, the sum of candidates' weight w_j must not exceed capacity c.

$$\begin{aligned} max \sum_{j=1}^{n} \quad & p_j \cdot x_j \quad && \text{subject to:} \\ \sum_{j=1}^{n} \quad & w_j \cdot x_j \leq c \quad && j = 1, \ldots, n \\ & x_j \quad \in \{0, 1\} \end{aligned} \tag{5.1}$$

The profit p_j composes of the benefit b_j less the modification costs $mcost_j$ (e.g., for index selection) [CFM95, LSSS07a, Lüb07, Lüb08]. Benefit b_j represents the advantage to take candidate j; whereas $mcost_j$ represents the costs to achieve benefit b_j .

$$\begin{aligned} p_j = \quad & b_j - mcost_j \quad \text{subject to:} \\ j = \quad & 1, \ldots, n \end{aligned}$$

In the following, we adapt the general Knapsack problem to the database tuning domain. We define capacity c as resource constraint rc to be more consistent to the domain. That is, rc describes the maximum available resources to compute candidate j (i.e., tasks/queries). The weight w_j, which we denote as demand d_j in the following, describes the proportionally consumption of the maximum available resource rc. We state that demand d_j and resource constraint rc describe the same measure (e.g., I/O). We present our derived Knapsack problem in Equation 5.2.

$$\begin{aligned} max \sum_{j=1}^{n} \quad & (b_j - mcost_j) \cdot x_j \quad && \text{subject to:} \\ \sum_{j=1}^{n} \quad & d_j \cdot x_j \leq rc \quad && j = 1, \ldots, n \\ & x_j \in \quad \{0, 1\} \end{aligned} \tag{5.2}$$

Until now, we do not compute the profit of different storage architectures with our derived Knapsack. Therefore, we introduce variable i that describes the corresponding storage architecture[4]. That is, we have to select candidates j from i classes with $i \in \{CS; RS; \ldots; m\}$ now. Consequently, we have to solve a multiple-choice Knapsack [SZ79] to select the optimal storage architecture. According to Sinha and Zoltners [SZ79] and in tandem with our previous considerations, we obtain a derived

[4]In the following formulas, we abbreviate Column Stores as CS and Row Stores as RS.

multiple-choice Knapsack problem. We present our (abstract) DM in Equation 5.3.

$$\begin{aligned} max \sum_{i}^{m} \sum_{j=1}^{n} & \quad (b_{ij} - mcost_{ij}) \cdot x_{ij} & \text{subject to:} & \\ \sum_{j=1}^{n} & \quad d_{ij} \cdot x_{ij} \leq rc_i & j = 1, \ldots, n & \\ x_{ij} \in & \quad \{0,1\} & i \in \{CS, RS, \ldots, m\} & \end{aligned} \tag{5.3}$$

We point out, several implementations and optimizations for Knapsack problems are available (e.g., by Pisinger [Pis95]); hence, we do not consider these aspects in more detail. We derive advisor approaches from the multiple-choice Knapsack problem in Equation 5.3 for different application fields in the following sections.

We develop *three* variants of our DM in the following sections. First, we introduce the ***online DM*** based on linear programming in Section 5.2.1. We develop the online DM to support optimal architecture selection with direct DBMS-statistic extraction. Second, we introduce a (offline) ***design-prediction model*** that analyzes predicted (and/or) future workloads. As a result, we obtain an indication of the optimal storage architecture (cf. Section 5.2.2). Third, we combine the online and design-prediction DM to the ***offline benchmarking model***. This model selects the optimal architecture like the online DM, but copes with architecture selection on predicted workloads like the design-prediction model as well. We support sample and/or predicted workloads as well as extracted DBMS statistics in the third DM variant (cf. Section 5.2.3).

5.2.1. Online Analysis with Statistics from DBMS

For our *online DM*, we use extracted statistics directly from DBMS. Furthermore, we reuse query plans [ABC+76] provided by relational DBMS in any shape (also cf. Section 2.4). That is, we do not introduce new cost measurements. We argue that we obtain the best possible initial values for our computations based on direct DBMS-optimizer output (estimates) [FST88]. For the following considerations, we assume that statistics of different architectures (here – Row Store and Column Store) are provided to our online DM. That is, the extracted statistics are normalized and stored in the workload-pattern framework (cf. Chapter 4 for more details).

We select the optimal storage architecture based on statistics that represent a number of different cost estimates (e.g., CPU costs). Therefore, we have to decide which estimation we use for optimizations. We state that our DM is transparent to cost functions. Nevertheless, cost functions can be arbitrarily complex (cf. Section 5.1). On the one hand, a cost function can only represent one cost criterion (e.g., CPU costs). On the other hand, complex cost functions range from a combined cost function that takes two criteria into account (e.g., CPU and I/O costs) to very complex cost functions that take all available cost criteria into account (for more details cf. Section 5.1). However, we derive a simplified optimization problem

for the storage architecture decision in the following.

We derive our online DM from the abstract DM presented in Equation 5.3 (cf. Section 5.2). We simplify the abstract DM by *two* aspects. First, we invert the optimization criterion from maximum profit (benefit minus costs) to minimum costs for a workload. We argue that a priori decision causes a suggestion for either architecture, thus, we have (a) no modification costs ($mcost_{ij}$) for storage modifications and (b) no benefit (b_{ij}) compared to other architectures. In contrast, we compare workload costs for the architectures, thus, we suggest the architecture that causes the minimum costs. That is, we replace the benefit b_{ij} and the modification costs $mcost_{ij}$ directly by a cost function $C(i,j)$. Second, we omit the capacity criterion of the abstract DM, and thus, the resource constraint rc_i is not applicable. We note that DBMS statistics may include operational information (e.g., current resource consumption), but for a priori decisions, we use only extracted DBMS statistics that are independent from operational statistics. Our decision shall be independent from current state of OS and/or DBMS. Furthermore, we assume a representative given workload for our analysis. In summary, we suggest the optimal storage architecture for a given workload without any resource constraints.

We build the architectural decision upon a linear program (cf. [Chv83, Pages 341 ff.]) that is related to the assignment problem[5] [Mun57]. We define a cost function (i.e., execution costs) $C(i,j)$ according to the previous discussed optimization-criterion inversion. We note that the cost function $C(i,j)$ has – in contrast to profit p_{ij} – no constraints on resource consumption rc_i. Furthermore, we define a (database) task as a part of a query (e.g., `ID1` represents a join operation within query `Q15`). We represent this database task within a workload as ID 15.1 according to the definition in Section 4.3 (cf. Figure 4.12). The assignment x_{ij} is constrcuted by the set of database tasks T and the storage architecture $\{CS; RS\}$. We assume that cost function values $C(i,j)$ exist for a task j and a storage architecture i. Consequently, we set up the derived online DM in Equation 5.4.

$$min \sum_{i \in \{CS;RS\}} \sum_{j \in T} C(i,j) \cdot x_{ij} \quad \text{subject to:}$$

$$\begin{aligned} \sum_{j \in T} x_{ij} = & \begin{cases} |T| \\ 0 \end{cases} & \forall i \in \{CS; RS\} \\ \sum_{i \in \{CS;RS\}} x_{ij} = & \quad 1 & \forall j \in T \\ x_{ij} \in & \quad \{0,1\} & \forall i \in \{CS; RS\}, \forall j \in T \end{aligned} \tag{5.4}$$

Our online DM has *two* constraints. First, we have to ensure that either all or none of the tasks are performed by either architecture because we suggest the one architecture with minimal costs for a given workload. Therefore, we define the

[5]The assignment problem is another specific problem of the combinatorial optimization.

domain of i subject to $\forall i \in \{CS; RS\}$. The cardinality of T ($|T|$) represents the number of tasks in a workload. In consequence, we obtain an alternation of task executions by the case distinction of $|T|$ and 0 (cf. Equation 5.5). That is, whenever x_{ij} is not zero for one value of i then x_{ij} have to be zero for all other values of i.

$$\sum_{j \in T} x_{ij} = \begin{cases} |T| \\ 0 \end{cases} \qquad \forall i \in \{CS; RS\} \tag{5.5}$$

Second, the binary variable $x_{ij} = 1$ states that we chose an item j from the class i. That is, x_{ij} can be only 0 or 1 and the sum of x_{ij} over i have to be 1, thus, $x_{ij} = 1$ is allowed for one combination of i and j only whereas $x_{ij} = 0$ for all other combinations of i and j. Consequently, $\sum_{i \in \{CS;RS\}} x_{ij} = 1 \; \forall j \in T$ guarantees that we execute all tasks T exactly once (cf. Equation 5.6).

$$\begin{aligned} \sum_{i \in \{CS;RS\}} x_{ij} = \quad & 1 \qquad \forall j \in T \\ x_{ij} \in \quad & \{0,1\} \quad \forall i \in \{CS; RS\}, \forall j \in T \end{aligned} \tag{5.6}$$

We state that our online DM (cf. optimization problem in Equation 5.4) suggests the storage architecture with guaranteed minimal costs taking into account both constraints and a previously defined cost criterion. However, we have to enlarge the co-domain of i only to extend our approach for other architectures (e.g., PAX [ADHS01] or SPAX [Bößß09] systems) because our DM is transparent to different architectures. We argue, architectures have to comply *two of following three* conditions to apply our approach with other architectures:

(a) The architectures to be compared have to support semantically the same workload (e.g., the same data model) as well as

(b_1) cost estimates for data processing have to be available (e.g., optimizer output) or at least

(b_2) provide sound cost samples for one of the following DM variants.

Furthermore, we assume for feasible (comparative) computations that at least two existing database systems and their workload statistics are given. Nevertheless, we can adapt the DM easily by changing the constraints and/or cost criteria (respectively cost functions). Hence, we can also use our approach for hybrid architectures in the future, that are not disjoint from their basic architectures (e.g., mixed column-row-store architecture). Moreover, we can identify very different tasks for different architectures by a sensitivity analysis. We outline our first results in Section 5.4, which we use to improve our framework.

We argue that the amount of queries and tasks may be unmanageable in practice, due to degenerated information content of a (infinite) workloads. Nevertheless, we are able to analyze the costs on different granularities (e.g., on sub-pattern or pattern

level; cf. Chapter 4). Therefore, our workload-statistic framework is able to group similar tasks and/or to restrict the number of tasks at an appropriate level. Moreover, we state that the set of system tasks T is not limited to one granularity level. That is, we define a task T by default as part of a query or as a database operation respectively, but we can use more coarse-grained query workload partitioning (cf. Chapter 4). The granularity of $C(i,j)$ has to match the granularity of tasks (cf. Section 5.1). That is, the input granularity as well as the computation granularity is selectable, thus, we obtain a maximum degree of freedom. Consequently, our linear program remains manageable for practical use. We argue, a high level abstraction contradicts the goals of the online DM. We do not consider uncertainty concerning to query structure in our online DM. In consequence, each query has already been analyzed and partitioned, thus, we derive the overall cost and are able to rebuild the query structure.

We summarize, the result of the linear program is the optimal storage architecture for a given workload according to the cost function $C(i,j)$. We state that our approach to select the optimal storage architecture is comparable to design advisors (e.g., by Zilio et al. [ZZL+04, ZRL+04]) that suggest the optimal physical design within systems. That is, our online DM suggest physical design between various systems (inter) and not within systems (intra) at this point. Furthermore, we suggest extensions that detect mislead design (e.g., as the design alerter by Bruno and Chaudhuri [BC05, BC06]). Therefore, we assume existing systems to enable a what-if analysis [BC05]. We face the necessity of existing systems with the following extensions (cf. Sections 5.2.2 and 5.2.3) to enable design alerts (i.e., what-if analysis) under uncertainty. Nevertheless, we assume that our model for optimal architecture selection assumes that systems do not change query structures which build upon all database queries and their corresponding execution information. That is, query-structure changes causes different costs, thus, we have to compute architecture selection with different cost values. In conclusion, we suggest a sensitivity analysis that evaluates the restrictions (e.g., granularity vs. uncertainty) of our model and examines the cost function especially in a more fine-granular way (e.g., multi-object optimization [MA04]). We present the selection procedure in detail in Section 5.3.

5.2.2. Offline Design Prediction

We showed the decision of optimal architecture for a workload based on real DBMS statistics in the previous section. We argue, real DBMS-statistics are mostly not available for predictions for architectural design. Consequently, we adapt the online DM from Section 5.2.1 to estimate the optimal storage architecture without DBMS-provided statistics. The adapted DM identifies the optimal storage architecture for predicted workloads. We consider predicted workloads as samples from existing systems; or as future workloads that are computed artificially (maybe arbitrarily) to discover future challenges. For predicted workloads, we assume that the structure of queries or workload itself is unknown; that is, we have to consider uncertainty of cost estimation for our use case. However, our DM in fact is independent from query

structures but it is not independent from the query-cost structure (cf. Chapter 3 and Section 4.3). That is, we have to estimate the query-cost structure for design predictions. Nevertheless, we have to consider the uncertainty of a Multi-Criteria Decision Problem (MCDP), due to the fact, different tasks within queries or workloads generate multi-dimensionality.

We present an approach that selects the optimal storage architecture based on DBMS-statistics, and we consider uncertainty of artificial cost estimates (i.e., for predicted workloads); that is, we only have to join both ideas to predict architectural design. Hence, we are able to overcome a drawback of the online DM – the necessity of existing systems – if we are able to compute predicted workloads. Therefore, we introduce the (offline) *design-prediction model* in the following. We assume the same prerequisites as for the online DM. That is, we incorporate the predicted workload structure, associate the query-cost structure to our workload patterns, and include fraction on the overall workload (cf. Chapter 4).

First, we extend our previous problem (cf. Equation 5.4) for uncertainty consideration. We argue, an extension of the cost function $C(i,j)$ regarding uncertainty is sufficient and promising. On the one hand, we concatenate directly the uncertainty of cost estimation with the cost computation; and on the other hand, our adaptation is transparent to the previous optimization problem due to the fact that we replace $C(i,j)$ with a new cost function $C^*(i,j)$. Therefore, we use probability theory to represent uncertainty, and thus, to represent samples, respectively future workload, as well as changes in DBMS behavior. However, we need predicted workloads in both use cases, and hence, we combine both aspects in one cost function $C^*(i,j)$. We introduce a probability function $p(i,j)$ to represent predicted workloads. That is, we analyze a predicted workload – a set of database tasks (cf. Section 5.2.1) – according to the frequency of a certain task to figure out its probability (see Equation 5.7). Consequently, we derive the probability that a task j has to be computed in a workload, whereas the sum of all probabilities $p(i,j)$ is equal to 1, i.e., probability values $p(i,j)$ represent the ratio of a task j to the (predicted) overall workload which equals the set of tasks T_{WL}.

$$\sum_{j \in T_{WL}} p(i,j) = \quad 1 \quad \forall i \in \{CS; RS\} \tag{5.7}$$

Due to the (potential) high number of tasks, a partition for tasks as well as for $C(i,j)$ has to be done. We partition the tasks according to our workload structure T_{WL} (cf. Section 4.2). As an exemplary result, we use a task set $T_{WL} = \{Join, Tuple\ Operations, Aggregation\ \&\ Grouping\}$, whereas the elements of T_{WL} may be further refined (e.g., join pattern consists of non-vector-based and vector-based joins). The partitioning of tasks does not appropriately restrict the information space. Therefore, average costs (for a certain granularity of $C(i,j)$) must be estimated, due to the fact that the granularity of $C(i,j)$ has to match to the granularity of tasks (cf. Section 5.1). In combination with our consideration in Chapter 4 that the granularity of workload partitioning is selectable, we obtain a

maximum degree of freedom. In consequence, the cost function $C(i,j)$ from Equation 5.4 adapts to $C^*(i,j)$ in Equation 5.8. We introduce a probability function $p(i,j)$ to represent predicted workloads and combine our cost function $C(i,j)$ to a new cost function $C^*(i,j)$ that considers uncertainty sufficiently and transparently for our design prediction.

$$\begin{aligned} C^*(i,j) &= p(i,j) \cdot C(i,j) \quad wrt. \\ \sum_{j \in T_{WL}} p(i,j) &= 1 \qquad \forall i \in \{CS; RS\} \end{aligned} \tag{5.8}$$

Second, we have to integrate the new cost function $C^*(i,j)$ into our existing approach (see Equation 5.4), thus, we substitute the cost function $C(i,j)$ with $C^*(i,j)$ (see Equation 5.9). Whereas, T_{WL} can be at any granularity, thus, we denote a set of tasks as T here.

$$min \sum_{i \in \{CS;RS\}} \sum_{j \in T} C^*(i,j) \cdot x_{ij} \quad \text{subject to:}$$

$$\begin{aligned} \sum_{j \in T} x_{ij} &= \begin{cases} |T| \\ 0 \end{cases} \qquad \forall i \in \{CS; RS\} \\ \sum_{i \in \{CS;RS\}} x_{ij} &= 1 \qquad \forall j \in T \\ x_{ij} &\in \{0,1\} \quad \forall i \in \{CS; RS\}, \forall j \in T \end{aligned} \tag{5.9}$$

We highlight, the optimization problem in Equation 5.9 encapsulates and hides important information content for further observations (e.g., impact of task frequency). Moreover, we argue that the current representation also hides the ease of adaptation between our DMs. Consequently, we combine the derivation of $C^*(i,j)$ (cf. Equation 5.8) with the adapted problem (Equation 5.9) to our design-prediction model in Equation 5.10.

We summarize that our design-prediction model enables us to estimate the cost of unknown (predicted) workloads. For quality of our DM according to design predictions, we assume that the estimated cost function is sufficient. Therefore, knowledge of domain experts is required. However, the partitioning of workload and cost function enables a more sophisticated approach than guessed decisions. We obtain an expectation value and cost plan under the given probabilities of the workload tasks. Consequently, our approach supports the development of sufficient design rules and design heuristics. This can be used for a sensitivity analysis where more restriction values are considered. The integration of heuristics results in a heuristic-impelled design by the DM. Moreover, we allow design prediction on different granularities of (input) information as well as on different quality of information content. These extensions transforms our previous approach (cf. Section 5.2.1) into a (heuristic)

design advisor, thus, we argue that side conditions hold validity. That is, we solve a non-weighted minimization problem that concerns only costs but no profit and is transparent to additional architectures and cost functions. Additionally, we suggest that the design-prediction model is – beside the advisor functionality – suitable to serve as design alerter for misdirected (or outdated) designs. In consequence, we state that our DM is easily adaptable or can be iteratively improved. In addition, we suggest the integration of user preferences. We integrate abstractly user preferences in the following Section 5.2.3.

$$min \sum_{i \in \{CS;RS\}} \sum_{j \in T} p(i,j) \cdot C(i,j) \cdot x_{ij} \quad \text{subject to:}$$

$$\begin{aligned}
\sum_{j \in T} x_{ij} &= \begin{cases} |T| \\ 0 \end{cases} & \forall i \in \{CS; RS\} \\
\sum_{i \in \{CS;RS\}} x_{ij} &= 1 & \forall j \in T \\
x_{ij} &\in \{0,1\} & \forall i \in \{CS; RS\}, \forall j \in T \\
\sum_{j \in T} p(i,j) &= 1 & \forall i \in \{CS; RS\}
\end{aligned} \tag{5.10}$$

5.2.3. Offline Benchmarking of Different Systems

We presented two approaches to select the storage architecture in the previous two sections. First, we selected the optimal storage architecture from (at least) two existing systems, and second, we selected the storage architecture under uncertainty using samples. Our second offline DM – the *offline benchmarking model* – combines and adopts ideas of the two previous approaches (cf. Section 5.2.1 and 5.2.2). That is, we integrate (a) the capability to benchmark two or more database systems from the online DM and (b) the capability to select the optimal storage architecture based on samples. In contrast to the online DM, we have no access to DBMS statistics, thus, we have to use workload samples (i.e., statistic samples). We focus our research on storage-architecture optimization to support (a priori) storage advisor, benchmark different architectures, and encourage hybrid-store development for mixed workloads (cf. Chapter 3). However, we do not focus on optimization of certain input variables for the decision problem (e.g., sample workload or workload estimation). In other words, we assume that workload samples are well-defined and sufficient (cf. Sections from 5.1 to 5.2.2 for more details). For further literature on workload aggregation, we refer the reader to query merging (e.g., [GK10]), query matching (e.g., [ZZL+04]), or workload-prediction approaches (e.g., for data center [GCCK07], online classification [HGR09]). Furthermore, we use a multi-criteria approach to evaluate system requirements and support user-preference integration with respect to workloads.

We introduce the offline benchmarking model in the following. We assume the

same prerequisites as for the design-prediction model. That is, we use the predicted workload structures (i.e., samples), associate the query-cost structures to our workload patterns, and include fraction on the overall workloads (cf. Chapter 4). Furthermore, we argue that granularity of costs (from samples) has to match to granularity of tasks (cf. Section 5.1). Nevertheless, we compute a MCDP, due to the fact; we benchmark systems with respect to different cost criteria (cf. Sections 5.2.1 and 5.2.2). Moreover, we have no access to environmental conditions concerning the systems to be benchmarked. That is, we have to consider user preferences to represent the desired system behavior (e.g., resource consumption, bottlenecks) – more general the systemic environment. Consequently, we compute a MCDP under uncertainty with respect to given (user) preferences.

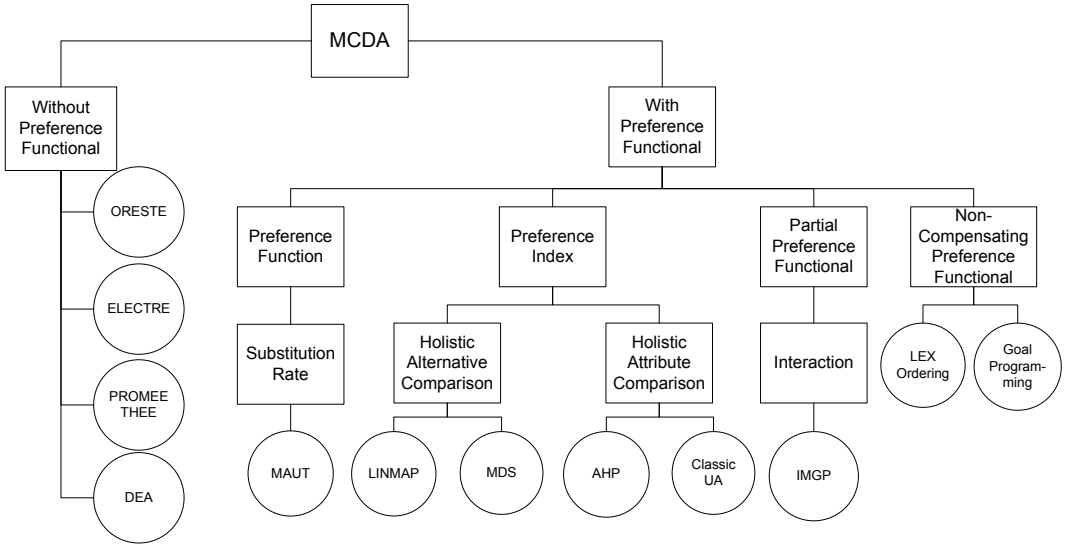


Figure 5.3.: *Classification of MCDA methods [BK10] according to Schneeweiß [Sch91]*

An important aspect in the context of Multi-Criteria Decision Analysis (MCDA) and uncertainty is the representation of the ranking function. Schneeweiß classifies MCDA methods according to ranking functions [BK10, Sch91] (cf. Figure 5.3) – for further applications of MCDA see [FGE05]. We argue, we have to consider the following *two* aspects to figure out a sufficient MCDA method according to Schneeweiß [Sch91]. *First*, we consider if an order of alternatives is possible and/or required. Schneeweiß classifies MCDPs that have an order of alternatives as MCDA methods with functional preference (cf. Figure 5.3). We argue that alternatives in terms of storage architectures have an explicit order, due to the fact that our computation of storage-architecture selection is based on (execution) costs, which are in algebraic order. According to von Winterfeldt and Edwards [vWE86], we are able to use Multi-Attribute Utility Theory (MAUT) whenever cost values apply and a utility function is available – for more background literature to MAUT, we recom-

mend [vNM53, Fis67a, Fis67b, KR76] to the reader. *Second*, we consider the type of order. Whenever substitution rules between alternatives are available, we assume that a preference function is constituted by these substitution rules. Furthermore, the alternatives have to be assigned to an interval measurement that represents the degree of difference and has an arbitrarily-defined zero point. We recognize that cost values according to storage architectures have no arbitrarily-defined zero point, due to the fact that costs are always positive values (0 or more). Nevertheless, we compute utility values of an alternative instead of alternative's minimum costs in the offline benchmarking model. We emphasize, the utility value – as quality of an alternative – is the distinctive feature concerning the two previous approaches (cf. Sections 5.2.1 and 5.2.2). If we would combine low costs of a task with high weights for the utility value; the resulting utility value puts such tasks in worse position compared to tasks with higher costs and lower weight. Thereby, low costs of a task represent good performance on the alternative, and weight for the utility value represents task's significance. In consequence, we would contradict the expected result – the most promising alternative. Therefore, we define the utility value as difference of alternative's cost, due to the fact that the difference of alternative's cost depicts the most feasible benefit.

We argue that alternatives' utility value can be positive and negative with respect to another alternative (cf. Equation 5.11). Moreover, the alternative's utility value is exactly zero – which we define as arbitrarily-defined zero point – whenever the alternatives to be benchmarked have the same costs. In consequence, MAUT allows us to quantify – how desirable is a certain alternative. That is, we are able to use MAUT in recommendation systems whenever a function (or estimation) of user preferences is present.

$$\begin{aligned} value_j(ALT_x) &= cost_j(ALT_y) - cost_j(ALT_x) \\ value_j(ALT_x) &< 0, \text{ if } cost_j(ALT_x) > cost_j(ALT_y) \\ value_j(ALT_x) &> 0, \text{ if } cost_j(ALT_x) < cost_j(ALT_y) \\ value_j(ALT_x) &= 0, \text{ if } cost_j(ALT_x) = cost_j(ALT_y) \\ &\quad \text{with } j \in T \text{ and } x,y \in i \end{aligned} \tag{5.11}$$

We summarize that we only use costs to compute the optimal storage architecture (i.e., the optimal alternative), due to the fact that no benefit b_{ij}, no modification costs $mcost_{ij}$, and no resource constraint rc_{ij} exist as well as for the previous two approaches (cf. Sections 5.2.1 and 5.2.2). We compute a set of attributes T (with $j \in T$) on a number of alternatives i – in our example Alternative x (ALT_x) and Alternative y (ALT_y). We define the set of attributes T as a group of database tasks (cf. Sections 4.2 and 5.2.1). Hence, we define the profit of x for task j – the utility value $value_j(ALT_x)$ – as difference between $cost_j$ for ALT_y and $cost_j$ for ALT_x (cf. Equation 5.11). Furthermore, we use weights for a task j ($weight_j$) to qualify user preferences. The sum of all weights according to a set of tasks T equals 1 (cf. Equation 5.12). In consequence, we achieve a MCDP that we can solve with MCDA

methods. According to Schneeweiß, von Winterfeldt, and Edwards [Sch91, vWE86], we can solve such MCDP with MAUT (cf. Equation 5.14).

$$\sum_{j=1}^{n} weight_j \quad = \quad 1, \text{ with } j \in T \tag{5.12}$$

$$u(P) \quad = \quad \sum_{j=1}^{M} u_j(P_j) \tag{5.13}$$

We recommend an alternative ALT_i (i.e., an architectural design) in a multi-attribute scenario for a set of tasks T according to its utility value $value(ALT_i)$. We are able to recommend the optimal architecture, due to the fact that a measurement between different dimensions is possible. We argue that the dimensions are defined by the workload differentiation from Chapter 4 depicted in Figure 4.10 (Page 41. We combine the general MAUT method according to Fishburn [Fis67a, Fis67b][6] (cf. Equation 5.13) with our previous considerations (cf. Figure 5.3, Equations 5.11 and 5.12) to the offline benchmarking model. In conclusion, the overall utility-value function is depicted in Equation 5.14.

$$\begin{aligned} value(ALT_i) \quad &= \quad \sum_{j=1}^{n} weight_j \cdot value_j(ALT_i) \text{ with} \\ \sum_{j=1}^{n} weight_j \quad &= \quad 1 \text{ and } j \in T \end{aligned} \tag{5.14}$$

We outline, we evaluate alternatives (ALT_i) – a set of DBMSs (e.g., but not limited to Column Store and Row Store) – by a utility-function *value* that takes the cost structure into account and weights all function values according to the workload structure. That is, we are able to differentiate between architectures (e.g., Column Store and Row Store), and additionally, we are able to benchmark different DBSs. We emphasize that the result of the utility function in Equation 5.14 is the utility of *one* alternative concerning to a second alternative. Hence, we recommend that alternative that has the higher utility value $value(ALT_i)$[7], thus, we solve in contrast to the two previous approaches a maximization problem – $max \sum_{i=2}^{m} value(ALT_i)$. In detail, we compare $m-1$ times to rank m alternatives, thus, we compute $m-1$ passes. If we compute a negative utility value then we recommend the compared alternative (i.e., $value_j(ALT_y)$ whenever $value_j(ALT_x) < 0$ with $cost_j(ALT_x) > cost_j(ALT_y)$).

Another advantage of MCDA with respect to MAUT is the derivation of user's

[6] We refer to the generalized additive independence (GAI) representation – for proofs and detailed explanations cf. [Bra12].

[7] We compare alternatives pairwise, thus, we do not have an overall result.

workload preferences. Hence, we develop a model of user preferences with respect to our workload hierarchy and to estimate the desired workload structure. We assume that weights include user preferences whenever user preferences shall be considered for alternative selection. Using the MAUT methodology, we obtain a recommendation or ranking between the set of alternatives. That is, we are able to perform a decision on this MCDP. Finally, our approach enables us to use this DM to figure out the optimal design for databases. We determine the offline benchmarking model as variant of the design-prediction model, thus, we once more emphasize the capability to advise storage-architecture design as well as the capability to be alerter in existing environments (e.g., other alternatives show significant better and sound results).

5.3. Evaluation

We present our DM as a priori storage-architecture-selection approach (cf. Section 5.2) and its assumptions and prerequisites (cf. Section 5.1). In this section, we show the proof of concept for our DM and present a case study based on the TPC-H benchmark [Tra10] with scale factor 1 (cf. Sections 5.3.1 and 5.3.2). We focus on the online DM (cf. Section 5.2.1), although the case study is adaptable for both other (offline) DMs (cf. Sections 5.2.2 and 5.2.3) as well. The process of evaluation is equivalent to the one for online DM. That is, we adapt the granularity of workload statistics from our workload patterns only. The online DM computes the optimal storage-architecture on a high degree of detail, whereas the offline DMs compute on more coarse-grained statistic granularity or even on samples only. However, our DMs support inter alia an efficient database (storage) design in context of optimal architecture concerning a workload (e.g., Row Store or Column Store).

For readability of the following considerations, we select two DBMSs only that are used in real-world OLAP applications – Infobright ICE 3.3.1 and Oracle 11gR2. Both DBMSs are installed with standard parameters, due to the fact we want to achieve sound experimental results without preconfigured optimizations (e.g., precomputed joins) for benchmarks. We loaded both schemata with standard DDL from the TPC-H benchmark. Moreover, we state that parameter (or index) configurations are not comparable across different DBMSs or different architectures. Note, Oracle automatically creates primary/foreign key indexes whereas ICE processes without indexes. Nevertheless, ICE utilizes meta information from the Knowledge Grid [Inf11b] that corresponds somehow to index-like access (cf. also Section 4.3). We make one exception – we restrict available main memory to 250 MB, thus, the TPC-H schema using 1 GB raw data does neither fit into RAM nor the fact table (`LINEITEM`). We choose this setup because we do not focus on main-memory processing – there are more suitable approaches (e.g., [Pla09, KN11]) – but we choose an evaluation setup for our DM that applies for most real-world DBSs – including disk access (i.e., more data than fits into RAM). We run our test series on a Samsung X65 with 2.2 GHz and 2 GB RAM using Ubuntu 10.04.1LTS.

We restrict information content to three queries from the TPC-H benchmark for

traceability of our considerations, and thus, save comprehensibility. We select the previous discussed queries `Q6`, `Q15`, and `Q16` to show different types of queries (which showed interesting results already; cf. Chapter 3 [Tra08]). Furthermore, we argue that these queries are representative to show the impact of architecture to performance – cf. Listings 6.3, A.26, and A.27 [Tra10]. For the interested reader, we attach the experimental results for all TPC-H queries in Appendix A.2. In the following, we select the optimal storage architecture for two different evaluation criteria: (a) the number of rows to be accessed and (b) I/O costs. However, the evaluation criterion as well as the cost function can be easily replaced or combined as described in Section 5.2.1.

We show the statistics for query `Q6`, `Q15`, and query `Q16` – gathered from ICE and Oracle – in the following section. Subsequently, we compute the storage-architecture decision for these queries concerning to two different cost criteria in Section 5.3.2.

5.3.1. Gathered Statistics from Workload Patterns

We extract the statistics from the corresponding optimizer – Oracle and ICE – as described in Chapter 4 [LKS11b, LKS11c]. Furthermore, we store the cost information in our workload patterns (cf. Section 4.3). We summarize the extracted cost information in Tables 5.1 and 5.3 to ensure readability. We use two cost measures – the optimization criteria – to emphasize that our approach is transparent to optimization criteria, and thus, to cost functions. That is, we replace the cost function transparently whenever we modify the optimization criterion – which can be arbitrarily complex. Furthermore, one cost measure is not sufficient to select the optimal storage architecture (cf. Sections 5.1 and 5.2.1). Additionally, we underpin our statements from Chapter 3 that two (or more) cost measures do not correlate across different architectures even though these cost measures are interdependent. Therefore, we select the accessed number of rows and the I/O cost to recommend the storage architecture (i.e., Oracle or ICE) for our exemplary workload – TPC-H queries `Q6`, `Q15`, and `Q16`. We note, ICE does not access single values of a column but accesses so-called data packs – compressed storage units in ICE. These data packs contain 65,536 values [Inf11b], thus, the values for ICE (number of rows) in Table 5.1 are multiples of the data-pack size[8].

We identify query operations by `IDs` that we extract from optimizers. Hence, we sustain processing schemes of queries and reuse these IDs in our workload patterns. That is, we are able to restore the processing scheme according to these IDs. In detail, we represent the first query operation by the highest ID and represent the last query operation by `ID0`. We assign jointly operation costs and operation IDs to the workload patterns, and therefrom we derive cost representation in Tables 5.1 to 5.5. This representation correlates to query-execution plans. We also use this methodology for administrative needs – for more details cf. Section 4.3.

[8] Note, we show for ICE all column values that have to be evaluated. The optimizer may suppose fewer values to be read.

Workload	**Q6**		**Q15**		**Q16**		$\sum$	
Pattern	**Oracle** (22.64sec)	**ICE** (2sec)	**Oracle** (26.28sec)	**ICE** (2sec)	**Oracle** (3.93sec)	**ICE** (1sec)	**Oracle**	**ICE**
Data Access	*ID2*: 155,900	*ID5*: 6,029,312 *ID4*: 6,029,312 *ID3*: 6,029,312 *ID2*: 6,029,312	*ID8*: 10,000 *ID6*: 218,657	*ID6*: 6,029,312 *ID4*: 6,029,312 *ID2*: 262,144 (4*65,536)	*ID9*: 800,000 *ID8*: 30,515 *ID6*: 500	*ID6*: 65,536 *ID5*: 851,968 *ID4*: 786,432	*1,215,572*	*38,141,952*
Non-vector			*ID1*: 20,000	*ID1*: 131,072	*ID7*: 830,515 *ID5*: 121,371	*ID3*: 1,114,112	*971,886*	*1,245,184*
Group By			*ID5*: 218,657	*ID5*: 225,954 *ID3*: 225,954	*ID4*: 114,828 *ID2*: 114,828	*ID2*: 473,096 (4*118,274)	*448,313*	*925,004*
Sort			*ID7*: 10,000 *ID4*: 10,000 *ID2*: 10,000		*ID1*: 15,000	*ID1*: 73,256 (4*18,314)	*45,000*	*73,256*
Sum	*ID1*: 155,900	*ID1*: 114,160					*155,900*	*114,160*
Projection	*ID0*: 1	*ID0*: 1	*ID3*: 10,000 *ID0*: 10,000	*ID0*: 4 (4*1)	*ID3*: 114,828 *ID0*: 15,000	*ID0*: 73,256 (4*18,314)	*149,829*	*73,261*

Table 5.1.: *Accessed rows (Oracle) respectively number of values for a column (ICE) for TPC-H queries* `Q6`*,* `Q15`*, and* `Q16`*.*

We start our observation with the number of rows to be accessed for a query. That is, we refer the data-access pattern respectively data to be read from disk. We show – in Table 5.1 – the number of rows to be accessed and the accessed number of column values[9] for the TPC-H queries `Q6`, `Q15`, and `Q16`. We remind that Oracle is a Row Store, thus, Oracle always accesses entire tuples. In contrast, ICE only accesses necessary columns, and thus, the required values. In Table 5.1, we observe this effect for query `Q6`. Oracle accesses the required columns in one pass, thus, it scans the `LINEITEM` table – the fact table – only once. The `LINEITEM` table has approximately six million tuples; due to predicate selection, Oracle reads 155,900 tuples only – cf. `ID2` for `Q6`. We direct the attention to ICE and we observe that ICE accesses the `LINEITEM` table in three passes, due to the fact that three columns (cf. `ID2,ID3,ID4`)[10] have to be processed for `Q6`. Moreover, ICE uses late materialization and does not reconstruct tuples before join execution. Nevertheless, we figure out that ICE in our test setup accesses more values than Oracle. Oracle reads 2.494.400 values (i.e., 155,900 rows with 16 columns) for `Q6` and ICE reads 24,117,248 values (i.e., 6,029,312 values for each processed column – 6,001,215 values allocate 92 data packs)[11] for `Q6`.

We observe akin results for `Q15`. That is, Oracle reads 3,568,512 values – i.e., 218,657 rows through view `REVENUE0` with 16 columns on base relation plus 10,000 rows on `SUPPLIER` with seven columns. At the same time, we observe 12,320,768 read values by ICE – i.e., four columns on `SUPPLIER` with one data pack each and two columns on `LINEITEM` with 92 data packs each.

In contrast, we see that ICE in total reads fewer values (i.e., 1,703,936) for `Q16` than Oracle (i.e., 4,278,135). We take a closer look to `Q16` (cf. Figure A.27 for the SQL representation). At first glance, we may conclude akin for `Q16` as for `Q6` and `Q15` according to the results in Table 5.1. At a closer look, we observe that ICE reads more values on `SUPPLIER` and `PART`; however, ICE reads fewer values for `PARTSUPP` because only `ps_suppkey` is scanned (see Table 5.2). Note, ICE takes advantage of subsequent column selectivity instead of reading each column independently `PART` (i.e., implicit tuple reconstruction). We argue, 786,432 values is a worst-case estimation as ICE reads all columns completely. Due to the selectivity of the respective predecessor columns, we argue that ICE reads fewer than 12 (estimated) data packs for `PART`. Moreover, ICE takes most advantage of the query structure. That is, ICE is able to compute the sub-query `supplier` as well as the `PARTSUPP` part by usage of one column each only. We observe that in fact, ICE scans only the columns `ps_suppkey` – the key of `PARTSUPP` – and `s_comment`. We summarize – Column Stores access tables multiple times whenever query processing involves several columns, and thus, Column Stores access inherently more values – this observation underpins previous considerations (cf. Section 3.2). Moreover, we represent for ICE the column values to be evaluated due to the absence of indexes in ICE. We state, the final number of

[9] For a homogenized summary of all pattern, we refer to Section 5.3.2.
[10] `l_shipdate`, `l_discount`, and `l_quantity`.
[11] ICE accesses all approximately six million values for each of the three `LINEITEM` columns.

TPC-H relation	Oracle	ICE
supplier	500 rows ∗ 7 columns	65,536 rows ∗ 1 column
partsupp	800,000 rows ∗ 5 columns	851,968 rows ∗ 1 column
part	30,515 rows ∗ 9 columns	≈ 262,144 rows ∗ 3 columns
$\sum$	***4,278,135*** values	***1,703,936*** values

Table 5.2.: *Accessed values of TPC-H Q16 per relation for ICE and Oracle.*

```
1 SELECT o_orderpriority,COUNT(*) AS order_count FROM orders
2 WHERE o_orderdate >= date '1993-07-01'
3 AND o_orderdate < date '1993-07-01' + interval '3' month AND EXISTS (
4    SELECT * FROM lineitem WHERE l_orderkey = o_orderkey AND l_commitdate < l_receiptdate)
5 GROUP BY o_orderpriority ORDER BY o_orderpriority;
```

Figure 5.4.: *TPC-H query Q4 [Tra10].*

accessed values the optimizer approximates is lower than we present in Table 5.1.

We showed that neither Column Stores are always advantageous nor Row Stores are. According to our first cost criterion, we observe divided results. We observe fewer values to be read for `Q6` and `Q15` by Oracle; in contrast, `Q16` gives an advantage to ICE. Hence, we consider our second cost criterion in the following. Note that we apply our statistic-normalization approach (cf. Section 4.4) for the following considerations. We highlight that ICE uses aggressive compression algorithms. We figure out that the C/R for the TPC-H benchmark in ICE is approximately 5 : 1. In our test setup, we measure 182 MB disk consumption for the 1 GB TPC-H data sets in ICE. For convenience, we do not separately examine the C/R for each column. One would usually expect that I/O costs increase by rising number of rows respectively values. However, we point out that physical data processing in DBMSs may differ from abstract data representation in the algebra. In general, direct (linear) correlation does not hold any longer between rows to be accessed and I/O costs – for Row Stores. We show the extracted cost information (i.e., I/O costs) in KBytes for our exemplary workload in Table 5.3.

```
1 SELECT SUM(l_extendedprice) / 7.0 AS avg_yearly FROM lineitem,part
2 WHERE p_partkey = l_partkey AND p_brand = 'Brand#23' AND p_container = 'MED BOX'
3 AND l_quantity < (
4    SELECT 0.2 * AVG(l_quantity) FROM lineitem WHERE l_partkey = p_partkey);
```

Figure 5.5.: *TPC-H query Q17 [Tra10].*

Workload	**Q6**		**Q15**		**Q16**		$\sum$	
Pattern	**Oracle** (22.64sec)	**ICE** (2sec)	**Oracle** (26.28sec)	**ICE** (2sec)	**Oracle** (3.93sec)	**ICE** (1sec)	**Oracle**	**ICE**
Data Access	*ID2*: 3,044.922	*ID5*: 4,382.416 *ID4*: 4,382.416 *ID3*: 4,382.416 *ID2*: 4,382.416	*ID8*: 703.125 *ID6*: 4,484.177	*ID6*: 4,715.582 *ID4*: 4,715.582 *ID2*: 410.051	*ID9*: 7,031.25 *ID8*: 1,222.769 *ID6*: 0.033	*ID6*: 102.513 *ID5*: 1,332.664 *ID4*: 1,230.152	*16,486.276*	*30,036.667*
Non-vector			*ID1*: 996.093	*ID1*: 205.026	*ID7*: 8,253.041 *ID5*: 5,935.107	*ID3*: 1,742.715	*15,184.241*	*1,947.741*
Group By			*ID5*: 4,484.177	*ID5*: 410.051 *ID3*: 410.051	*ID4*: 13,232.133 *ID2*: 5,494.699	*ID2*: 820.102	*23,211.711*	*1,640.204*
Sort			*ID7*: 703.125 *ID4*: 205.078 *ID2*: 292.969		*ID1*: 717.773	*ID1*: 410.051	*1,918.945*	*410.051*
Sum	*ID1*: 3,044.922	*ID1*: 4,382.416					*3,044.922*	*4,382.416*
Projection	*ID0*: 0.02	*ID0*: 102.513	*ID3*: 292.969 *ID0*: 996.093	*ID0*: 410.051	*ID3*: 13,232.133 *ID0*: 717.773	*ID0*: 410.051	*15,238.988*	*922.615*

Table 5.3.: *Accessed data of TPC-H queries* `Q6`*,* `Q15`*, and* `Q16` *in KBytes for Oracle and ICE.*

We remind that Oracle reads for query `Q6` and `Q15` fewer rows as well as values than ICE – 2,494,400 to 24,117,248 and 3,568,512 to 12,320,768 values, respectively. According to Table 5.3, we summarize the accessed data to 3,044.922 KBytes and 5,187.302 KBytes for Oracle; respectively for ICE, we summarize to 17,529.663 KBytes and 9.841,215 KBytes. That is, we observe a direct correlation between accessed rows respectively values and I/O cost. In contrast, we may observe the contradiction for `Q16` due to the fact that Oracle reads 831,015 rows and 8,287.222 KBytes whereby ICE reads 1,703,936 values and 3,587.943 KBytes. After detailed consideration (cf. Table 5.2), we highlight that the fact is actually inverse for `Q16`; Oracle reads more values than ICE – 4,278,135 to 1,703,936 values – resulting in higher I/O cost for Oracle namely 8,287.222 KBytes compared to 3,587.943 KBytes for ICE. That is, we observe a contradicted correlation between rows and I/O cost. Nevertheless, we argue the correlation holds for values and I/O cost. We consider the queries `Q4` and `Q17` from the TPC-H benchmark (cf. Listings 5.4 and 5.5) to show that `Q16` is not an artifact. Therefore, we observe 3,100,907 rows (i.e., 49,208,484 values) respectively 6,001,415 rows (i.e., 96,021,240 values) for Oracle versus 15,073,280 respectively 18,612,224 values for ICE (cf. Table 5.4) resulting in higher I/O cost for Oracle. That is, Oracle reads 67,074.205 KBytes and 82,053.135 KBytes whereby ICE reads 21,220.118 KBytes and 10,558.803 KBytes (cf. Table 5.5). We observe in our exemplary workload[12] that a system has not higher I/O cost that reads more rows but a system that reads more values. We conclude, the correlation is valid between values to be accessed and I/O cost for our exemplary workload, but the correlation is not valid across architectures (rows vs. values) without cost normalization. In summary, according to the results from Table 5.1 to Table 5.5, physical design based on basic heuristics or based on single cost measures is not sufficient for complex workloads. That is, we need decision support to select the optimal storage architecture as we propose in the preceding sections.

Please note, we only refer to the data-access pattern here – we show the overall comparison in Section 5.3.2. We argue that data of intermediate results is processed in main memory for the current setup. We may observe more complex effects on intermediate results for hybrid workloads (cf. Chapter 6) – in contrast to the current analytical (read-only) workload. Nevertheless, the remaining patterns characterize intermediate results (e.g., data size), and thus, we discover bottlenecks (e.g., disk swapping due to undersized main memory). Therefore, we show a summary of all workload patterns that occur in our exemplary workload assisted by our approach in the following section.

[12] We refer to Appendix A for further experimental results.

Workload	**Q4**		**Q17**		$\sum$	
Pattern	**Oracle** (27.29sec)	**ICE** (153sec)	**Oracle** (24.06sec)	**ICE** (1,495sec)	**Oracle**	**ICE**
Data Access	*ID4*: 3,042,903 (48,686,448 values) *ID3*: 58,004 (522,036 values)	*ID5*: 6,029,312 *ID4*: 6,029,312 *ID3*: 3,014,656 (2*1,507,328)	*ID6*: 6,001,215 (96,019,440 values) *ID5*: 200 (1,800 values)	*ID7*: 6,029,312 *ID6*: 524,288 (2*262,144) *ID5*: 6,029,312 *ID4*: 6,029,312	*9,102,322*	*18,612,224*
Non-vector	*ID2*: 3,100,907		*ID4* 6,001,415	*ID2*: 327,680	*9,102,322*	*327,680*
Tuple reconstruction				*ID3*: 12,320,768	–	*12,320,768*
Group By		*ID2*: 157,569 (3*52,523)			–	*157,569*
Sort	*ID1*: 58004	*ID1*: 10 (2*5)	*ID3*: 5,943 *ID1*: 5,943		*69,890*	*10*
Sum				*ID1*: 327,680	–	*327,680*
Projection	*ID0*: 5	*ID0*: 10 (2*5)	*ID2*: 5,943 *ID0*: 1	*ID0*: 1	*5,949*	*11*

Table 5.4.: *Accessed rows (Oracle) respectively number of values for a column (ICE) for TPC-H queries* `Q4` *&* `Q17`*.*

Workload	**Q4**		**Q17**		$\sum$	
Pattern	**Oracle** (27,29sec)	**ICE** (153sec)	**Oracle** (24.06sec)	**ICE** (1,495sec)	**Oracle**	**ICE**
Data Access	*ID4*: 65,374.869 *ID3*: 1,699.336	*ID5*: 8,488.048 *ID4*: 8,488.048 *ID3*: 4,244.024 (2*2122.012)	*ID6*: 82,047.861 *ID5*: 5.273	*ID6*: 3,420.457 *ID6*: 297.431 (2*148.716) *ID5*: 3,420.457 *ID4*: 3,420.457	*154,395.066*	*31,778.922*
Non-vector	*ID2*: 67,074.205		*ID4*: 82,053.134	*ID2*: 512.563	*67,074.205*	*512.563*
Tuple reconstruction				*ID3*: 29113.593	–	*29,113.593*
Group By		*ID2*: 0.3 (3*0.1)			–	*0.3*
Sort	*ID1*: 2,945.516	*ID1*: 0.2 (2*0.1)	*ID3*: 237.952 *ID1*: 75.448		*3,258.916*	*0.2*
Sum				*ID1*: 512.563	–	*512.563*
Projection	*ID0*: 0.254	*ID0*: 0.2 (2*0.1)	*ID2*: 237.952 *ID0*: 0.013	*ID0*: 0.1	*238.219*	*0.3*

Table 5.5.: *Accessed data of TPC-H queries* `Q4` *&* `Q17` *in KBytes for Oracle and ICE.*

5.3.2. Solution for the Optimization Problem in the Online DM

We show – in this section – a complete pass of the storage-advisor approach for our exemplary workload (`Q6`, `Q15`, and `Q16`) based on our decomposition approach for workloads (cf. Chapter 4) and our cost-estimation and advisor approach (cf. Section 5.2). The result is the selection of the optimal storage architecture.

We solve the assignment problem (cf. Section 5.2.1) for the architecture selection with a linear program based on the optimization problem in Equation 5.4. As linear programming language, we use A Mathematical Programming Language (AMPL) formulation [FGK02]. Due to the fact that we are interested in the optimal DBMS for a workload to a specific cost function, the problem is a mixed integer problem[13]. We present the AMPL-source code for our approach in Listing 5.6. The presented source code solves a minimization problem. According to the optimization, a selection of the cost values is necessary. In our example, two possible cost values (dimensions) are available. We can perform either an optimization according to the accessed rows (respectively values) or according to I/O cost. We note that – in practice – such simplification is not always feasible, often all cost influence structures have to be addressed. That is, workload decomposition is much more complex and additional DBMSs may be ranked. We highlight, our model is transparent to cost functions, thus, we easily adapt to arbitrary cost functions – cf. Section 5.2.2 where we introduce our approach with uncertainty (cf. Equation 5.8). We assume, the exemplary workload is defined by `Q6`, `Q15`, and `Q16` from the TPC-H benchmark[14]. We compute that ICE outperforms Oracle for the given workload (cf. Section 2.3 and Chapter 3). In the following, we present our DM based on ICE and Oracle according to two cost dimensions without loss of generality due to the fact that our approach is transparent to cost functions (cf. Section 5.1), cost measures, architectures and DBMSs, respectively (cf. Chapter 4).

First, the number of accessed rows respectively values have to be minimized. From our AMPL program, we figure out that Oracle accesses fewer rows ($\sum$ 2,986,500) than ICE accesses values ($\sum$ 40,572,817). Therefore, our DM will recommend Oracle with respect to the first cost dimension and without cost normalization across architectures. This result has been expected with respect to our results in Section 5.3.1 and the fact that rows are composed of an amount of values. However, we consider the normalized cost (i.e., values for both DBMS) and figure out that Oracle reads as well fewer values ($\sum$ 21,710,450) (from disk) than ICE. That is, we determine that surprisingly Oracle is optimal according to the first cost dimension for our workload even with cost normalization for the time being. Nevertheless, the ratio declines from approximately 1 : 13 (rows vs. values) to approximately 1 : 2 (values). We argue that architecture selection based on one cost measure (or dimension at all) – even normalized cost – is oversimplified (cf. Sections 5.1 and 5.2[15]).

[13] Mixed integer programming is the minimization or maximization of a linear function subject to linear constraints (cf. 5.2.1).
[14] We remind, this benchmark is designed to simulate OLAP workload.
[15] For additional information cf. Chapters 3 and 4.

```
1  set DBMS; # set of DBMSs for ranking
2  set WorkloadPattern; # set of Workload Patterns
3
4  param cost{i in DBMS, j in WorkloadPattern}; # cost
5  var assign{i in DBMS, j in WorkloadPattern}
6        binary; # = 1 if DBMS i is used, 0 otherwise
7  var use {i in DBMS} binary; # assignment that exactly one DBMS is used
8
9  minimize cost:
10   sum{i in DBMS, j in WorkloadPattern} cost[i,j]*assign[i,j];
11
12 subject to USAGE: sum{i in DBMS} use[i] = 1; # restriction that exactly one DBMS is in use
13 subject to Multi_Architecture {i in DBMS}:
14   sum {j in WorkloadPattern} assign[i,j] = 6 * use[i]; # this DBMS has to do all 6 Workload Pattern Tasks
15
16 subject to Tasks{j in WorkloadPattern}:
17    sum{i in DBMS} assign[i,j] = 1; # restriction that all tasks are performed
```

Figure 5.6.: *AMPL model for online decision – cost minimization.*

Workload	**Oracle** ($3 \times \oplus$)			**ICE** ($9 \times \oplus$)		
Pattern	**# rows**	**# values**	**I/O cost**	**# values**	**I/O cost**	
Data Access	*1,215,572*	*10,341,067* $\oplus$	*16,486.276* $\oplus$	*38,141,952*	*30,036.667*	
Non-vector	*971,886*	*2,219,770*	*15,184.241*	*1,245,184* $\oplus$	*1,947.741*	$\oplus$
Group By	*448,313*	*5,106,104*	*23,211.711*	*925,004* $\oplus$	*1,640.204*	$\oplus$
Sort	*45,000*	*166,000*	*1,918.945*	*73,256* $\oplus$	*410.051*	$\oplus$
Sum	*155,900*	*2,494,400*	*3,044.922* $\oplus$	*114,160* $\oplus$	*4,382.416*	
Projection	*149,829*	*1,383,109*	*15,238.988*	*73,261* $\oplus$	*922.615*	$\oplus$
$\sum$	***2,986,500***	***21,710,450***	***75.085,083***	***40.572.817***	***39,339.694***	

Table 5.6.: *Summary of accessed data (number of resp. KBytes) for Oracle and ICE concerning TPC-H queries Q6, Q15, and Q16.*

Second, we consider I/O-cost minimization for our workload. We assume that ICE induces less I/O cost due to aggressive compression, direct processing on compressed data, and vector operations (e.g., subsequent selectivity; cf. Chapter 3). Our AMPL program computes for Oracle 75,085.083 KBytes and for ICE 39,339.694 KBytes I/O cost. As we assume, the I/O cost is much lower for ICE than for Oracle, thus, we determine ICE as optimal solution according to our second cost dimension. We figure out additionally that the ratio for I/O cost between Oracle and ICE are the same as for accessed values, but now in vice versa direction (i.e., approximately $2:1$). We present the summary for our results in Table 5.6. We show the summation of cost per workload pattern and the total sum for each DBMS but not the constraints (i.e., subjects in Listing 5.6).

We determine that our model comes to a draw when we consider both cost dimensions and consider only totals for our exemplary workload. Moreover, we figure out that the cost ratios are contrary. That is, we observe for Oracle, the ratio is approximately $1:2$ according to the first and is approximately $2:1$ according to the second

cost measure (or vice versa from the point of view for ICE). However, we observe a draw in terms of optimizer estimation, but in terms of query execution we observe an explicit result on behalf of ICE for our sample workload (i.e., TPC-H queries `Q6`, `Q15`, and `Q16`). That is, ICE executes the queries in approximately 5 seconds whereas Oracle executes the queries in approximately 53 seconds (cf. Table 5.1 or 5.3)[16]. We argue that an evaluation of summarized results (e.g., Table 5.6) is (often) not sufficient for sound analysis of system behavior. Therefore, we suggest the evaluation of pattern among each other as well as against each other. A first coarse approach may be to count the number of workload pattern ("weighting") for each DBMS with lower cost in comparison. We remind, the distribution of accessed rows/values per workload pattern is available in Table 5.1 as well as we can see the distribution of data access in Table 5.3. Under inclusion of Table 5.6, we conclude that ICE claims 9 out of 12 workload patterns whereas Oracle claims 3 workload patterns. That is, we observe an advantage for ICE – which reflects better the measured query-execution times – instead of a draw. For more details we refer to Section 5.4.1, where we discuss weighting of cost and its evaluation.

In summary, we show a pass of our (online) storage-advisor approach with cost estimates from the query optimizers. We compare rows to be accessed with values to be accessed for Oracle (after cost normalization – cf. Section 4.4) and highlight the impact of cost normalization in Table 5.6. Moreover, we show the aggregated costs per workload pattern as well as total cost for each DBMS. We observe a draw between Oracle and ICE for values to be accessed and I/O cost according to total cost. We additionally suggest a simple voting approach to improve the decision due to the fact that cost aggregates for single workload pattern provide another conclusion. That is, ICE wins more workload pattern (less cost) than Oracle – 9 versus 3. However, we state that our model enables us to run a sensitivity analysis which identifies important cost drivers. Furthermore, it is possible to add easily more workload information which increases complexity for the decision makers. We also increase complexity by introducing more DBMSs. The obtained query-decomposition information on rows, cf. Table 5.1, and I/O cost, cf. Table 5.3, can be aggregated for each DBMS and workload pattern on arbitrary degree of detail. That is, we are able to control degree of detail for input information on arbitrary level as well as for decision processing in our approach. For our example workload, we assume that all three selected queries (`Q6`, `Q15`, and `Q16`) are executed in the same ratio. Otherwise, we have to adjust the cost structure by a ("weighting") function with respect to the query frequency (cf. Sections 5.2.2 and 5.2.3). We discuss "weighting" functionality in the following section.

5.4. Improvements for Decision-making Process

We recommend the optimal storage architecture for workloads based on extracted statistics a priori. Therefore, we use our design advisor that we present in Section 5.2.

[16]Please note, we do not tune the DBMSs configurations (cf. Section 5.3)

We show in Section 5.3 that unfiltered and non-enriched statistic information can induce unsatisfactory results. That is, we extend our idea – to count the winner per workload pattern – from Section 5.3.2 by weights to obtain better results. We present a more sophisticated approach in Section 5.4.1. Furthermore, we present design heuristics as by-product of the advisor approach in Section 5.4.2. Design heuristics are intended for use cases, whenever we have no detailed workload information. We point out that heuristics have no impact on the a-priori advisor approach due to the fact; in general a priori design decisions are not time-critical as query optimization is. We argue, improvements for our DM are independent from the core idea and form an independent module in the decision-making process (cf. Figure 5.7). Nevertheless, we show extensions of the core idea in Sections 5.2.2 and 5.2.3 that are reflected in the following. The (following) improvements are iteratively refined.

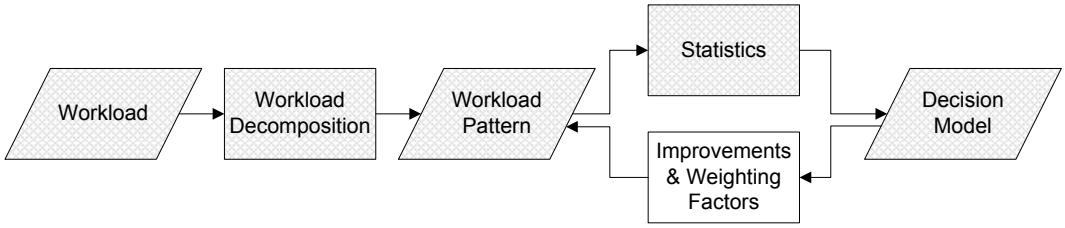


Figure 5.7.: *Classification for our improvements in the decision workflow.*

5.4.1. Weights for cost estimation

We show a pass of our storage-advisor approach for our example workload in Section 5.3.2. We discuss "weighting" functionality with respect to cost estimates for decision making. So far, we evaluate neither the impact of query frequency nor the impact of (user) preferences to our decision-making process (apart from our count-the-winner example). Nevertheless, now both aspects of "weighting" cost estimates are included in our approach. That is, we show integration of query frequency based on our (offline) design-prediction model (cf. Section 5.2.2) on the one hand. On the other hand, we show integration of preferences (i.e., weights) based on our offline benchmarking model (cf. Section 5.2.3).

First, we use our design-prediction model (cf. Section 5.2.2) to represent query frequency assuming that DBMS-provided statistics are available – which is no necessary condition for this model. In Section 5.3.2, we state that the queries in our sample workload are executed in the same ratio. Nevertheless, we transfer the information of tasks and their cost from linear program to the design-prediction model. We argue, the probability function $p(i,j)$ represents in absence of uncertainty nothing else than the execution frequency of task j (with $j \in T$; cf. Equation 5.10). We conclude that the overall number of tasks j for our sample workload is for Oracle 22 and for ICE 20. However, we shift the ratio of query execution for our exemplary workload

	exemplary Workload						
Workload	**Q6**		**Q15**		**Q16**		
Pattern	**Oracle**	**ICE**	**Oracle**	**ICE**	**Oracle**	**ICE**	$\sum$
Data Access	1 (1/22)	4 (1/5)	2 (1/11)	3 (3/20)	3 (3/22)	3 (3/20)	6 (3/11) / 10 (1/2)
Non-vector			1 (1/22)	1 (1/20)	2 (1/11)	1 (1/20)	3 (3/22) / 2 (1/10)
Group By			1 (1/22)	2 (1/10)	2 (1/11)	1 (1/20)	3 (3/22) / 3 (3/20)
Sort			3 (3/22)		1 (1/22)	1 (1/20)	4 (2/11) / 1 (1/20)
Sum	1 (1/22)	1 (1/20)					1 (1/22) / 1 (1/20)
Projection	1 (1/22)	1 (1/20)	2 (1/11)	1 (1/20)	2 (1/11)	1 (1/20)	5 (5/22) / 3 (3/20)
$\sum$	**3 (3/22)**	**6 (3/10)**	**9 (9/22)**	**7 (7/20)**	**10 (5/11)**	**7 (7/20)**	**22 (1/1) / 20 (1/1)**

Table 5.7.: *Frequency and relative share (in ( )) of tasks per pattern and per query in our exemplary workload.*

	shifted Workload						
Workload	**Q6**		**Q15**		**Q16** – *twice*		
Pattern	**Oracle**	**ICE**	**Oracle**	**ICE**	**Oracle**	**ICE**	$\sum$
Data Access	1 (1/32)	4 (4/27)	2 (1/16)	3 (1/9)	6 (3/16)	6 (6/27)	9 (9/32) / 13 (13/27)
Non-vector			1 (1/32)	1 (1/27)	4 (1/8)	2 (2/27)	5 (5/32) / 3 (1/9)
Group By			1 (1/32)	2 (2/27)	4 (1/8)	2 (2/27)	5 (5/32) / 4 (4/27)
Sort			3 (3/32)		2 (1/16)	2 (2/27)	5 (5/32) / 2 (2/27)
Sum	1 (1/32)	1 (1/27)					1 (1/32) / 1 (1/27)
Projection	1 (1/32)	1 (1/27)	2 (1/16)	1 (1/27)	4 (1/8)	2 (2/27)	7 (7/32) / 4 (4/27)
$\sum$	**3 (3/32)**	**6 (6/27)**	**9 (9/32)**	**7 (7/27)**	**20 (5/8)**	**14 (14/27)**	**32 (1/1) / 27 (1/1)**

Table 5.8.: *Frequency and relative share (in ( )) of tasks per pattern and per query in the shifted workload.*

– we execute query `Q16` twice as often as other queries[17]. In simplest construction, we execute each query *once* with exception of `Q16` that we execute *twice*. Hence, we recompute the overall number of tasks j for Oracle to 32 and for ICE to 27.

We summarize the execution frequency of tasks query-wise as well as pattern-wise in Tables 5.7 and 5.8. In comparison, we observe the expected shift in distribution of tasks. That is, the relative share of `Q16`-tasks rises from $5/11$ ($\approx 45\%$) to $5/8$ ($\approx 63\%$) for Oracle respectively from $7/20$ (35%) to $14/27$ ($\approx 52\%$) for ICE; whereas the relative share of other queries lower from $6/11$ ($\approx 55\%$) to $3/8$ ($\approx 38\%$) for Oracle and from $13/20$ (65%) to $13/27$ ($\approx 48\%$) for ICE. We observe the same distribution shift accordingly for patterns (e.g., share of the non-vector-based-join pattern for `Q16` (and for workload): $1/11 \approx 9\% \nearrow 1/8 \approx 13\%$ ($\approx 14\% \nearrow \approx 16\%$) for Oracle and $1/20 = 5\% \nearrow 2/27 \approx 7\%$ ($10\% \nearrow \approx 11\%$) for ICE). Hence, we observe also an effect for the cost estimates (cf. Table 5.6). We show the recomputed summary of accessed data for the shifted workload in Table 5.9[18].

[17]Note, we select a simple example consciously to ensure readability and traceability.

[18]Note, we assume that costs sum up further on for query processing due to missing knowledge about caching behavior, buffer management, query sequence, and so on.

Workload	**Oracle** ($3 \times \oplus$)			**ICE** ($9 \times \oplus$)		
Pattern	**# rows**	**# values**	**I/O cost**	**# values**	**I/O cost**	
Data Access	*2,046,587*	*14,619,202* $\oplus$	*24,740.328* $\oplus$	*39,845,888*	*32,701.996*	
Non-vector	*1,923,772*	*4,369,540*	*29,372.389*	*2,359,296* $\oplus$	*3,690.456*	$\oplus$
Group By	*677,969*	*6,713,696*	*41,938.543*	*1,398,100* $\oplus$	*2,460.306*	$\oplus$
Sort	*60,000*	*226,000*	*2,636.718*	*146,512* $\oplus$	*820.102*	$\oplus$
Sum	*155,900*	*2,494,400*	*3,044.922* $\oplus$	*114,160* $\oplus$	*4,382.416*	
Projection	*279,657*	*2,706,217*	*29,188.894*	*146,517* $\oplus$	*1,332.666*	$\oplus$
$\sum$	***5,143,885***	***21,710,450***	***130,921.864***	***44,010,473***	***45,387.942***	

Table 5.9.: *Summary of accessed data (number of resp. KBytes) for Oracle and ICE in the **shifted** workload.*

Concerning our results from Section 5.3.2, we conclude that the ratio between Oracle and ICE for the first criterion (number accessed values) remains approximately 1 : 2; whereas the ratio for the second criterion (I/O cost) shifts from approximately 2 : 1 to approximately 3 : 1. We state, the ratio change may lead to another result in certain environments/parameters as the results in Section 5.3.2 lead to. However, we focus on representation of query frequency in workloads which we show here. We just mention on the edge that our voting example (count-the-winner; cf. Section 5.3.2) remains 3 : 9 (cf. Table 5.10). Our query-frequency representation is capable for DBMS-provided statistics as well as for predicted workloads[19]. Furthermore, we argue that the query-frequency representation can be on arbitrary degree of detail (like cost representation; cf. Sections 4.2 and 4.4), thus, we are still able to compute the design decision on arbitrary degree of detail assuming that the degrees of freedom are compatible.

Second, we show the impact of weights to the design decision. Therefore, we utilize our offline benchmarking model (cf. Section 5.2.3). We compute the value of an alternative ($value(ALT_i)$; i.e., a system i) and compare the value pairwise; whereas the comparison is possible on arbitrary degree of detail (e.g., task j is an operation or a pattern; cf. Section 4.2). Note, we only show value computation for pattern and overall value of an alternative to save readability and comprehensibility. That is, a workload pattern is equal to a task j. We remind that the value of a task ($value_j(ALT_i)$) computes from difference of cost ($cost_j(ALT_i)$) which we show in Table 5.6 (on Page 86). We recompute the according example with our offline benchmarking model and show the results in Table 5.10. We use uniformly distributed weights ($weight_j$) for this computation (cf. Case A in Table 5.11) which is equivalent to unweighted computation. That is, $weight_j$ is $1/6$ for each pattern due to six workload pattern in total. We state that the result is comparable with the result in Section 5.3.2 – Oracle has higher I/O cost resulting in higher $value(ALT_{ICE})$ (i.e., 35,745.39) as well as Oracle accesses fewer values resulting higher $value(ALT_{Oracle})$ (i.e., 18,862,367). Even our voting example (count-the-winner; cf. Section 5.3.2) remains 3 : 9. However, we argue that the amount of $value_j(ALT_i)$ (i.e., benefit; cf.

[19] We assume, the query frequency is inherently included into predicted workloads.

Workload Pattern	accessed values $value_j(ALT_{Oracle})$		accessed values $value_j(ALT_{ICE})$		I/O cost $value_j(ALT_{Oracle})$		I/O cost $value_j(ALT_{ICE})$	
Data Access	*27,800,885*	⊕	*-27,800,885*		*13,550.391*	⊕	*-13,550.391*	
Non-vector	*-974,586*		*974,586*	⊕	*-13,236.500*		*13,236.500*	⊕
Group By	*-4,181,100*		*4,181,100*	⊕	*-21,571.507*		*21,571.507*	⊕
Sort	*-92,744*		*92,744*	⊕	*-1,508.894*		*1,508.894*	⊕
Sum	*-2,380,240*		*2,380,240*	⊕	*1,337.494*	⊕	*-1,337.494*	
Projection	*-1,309,848*		*1,309,848*	⊕	*-14,316.373*		*14,316.373*	⊕
$\sum value(ALT_i)$	*18,862,367*		*-18,862,367*		*-35,745.390*		*35,745.390*	

Table 5.10.: *Case A: Partial results for value(ALT_i) per pattern and cost function and overall value(ALT_i) per system and cost function.*

Workload Pattern	Case A Oracle	Case A ICE	Case B Oracle	Case B ICE	Case C Oracle	Case C ICE	Case D Oracle	Case D ICE
Data Access	1/6	1/6	3/11	1/2	1/10	1/10	1/20	1/20
Non-vector	1/6	1/6	3/22	1/10	9/50	9/50	9/40	9/40
Group By	1/6	1/6	3/22	3/20	9/50	9/50	9/40	9/40
Sort	1/6	1/6	2/11	1/20	9/50	9/50	9/40	9/40
Sum	1/6	1/6	1/22	1/20	9/50	9/50	9/40	9/40
Projection	1/6	1/6	5/22	3/20	9/50	9/50	1/20	1/20
$\sum weight_j$	1	1	1	1	1	1	1	1

Table 5.11.: *Weights per pattern for value(ALT_i) calculation concerning exemplary workload.*

Section 5.2.3) has impact on the overall $value(ALT_i)$; in contrast benefits are covered by votes in our voting example (cf. Section 5.3.2). Hence, we show the impact of weights ($weight_j$) to $value(ALT_i)$ in three other cases.

In Case B, we readopt the frequency idea from above and map execution frequency of tasks to weights. We take over the results (i.e., sum of relative share per pattern) from Table 5.7 and assign these as $weight_j$ to the corresponding pattern in Table 5.11. We observe that $value(ALT_i)$ changes for each pattern (per system and cost function; e.g., $value_{Group\ By}(ALT_{ICE}) declines : 21{,}571.507 \searrow 3{,}235.7260$) as well as for $\sum value(ALT_i)$ (e.g. $\sum value(ALT_{ICE}) : 35{,}745.39 \searrow -59.7935$; cf. Case B in Table 5.12). Nevertheless, we observe no change in ranking between $value(ALT_i)$ neither per cost function nor between cost functions themselves; whereas amount of $value(ALT_i)$ change as expected. That is, we observe higher $value(ALT_{Oracle})$ concerning accessed values (i.e., $6{,}456{,}263 > -12{,}855{,}692$) and lower $value(ALT_{Oracle})$ concerning I/O cost (i.e., $-4{,}518.256 < -59.793$) again (or vice versa from the point of view for ICE). For Case C and D (cf. Table 5.11), we use (artificial) user preferences to show the impact of user experience.

In Case C, we argue that data access has a small (or negligible) impact on the architecture decision (i.e., $weight_{Data\ Access}$ is 1/10); whereas the query process-

	Case B			
Workload	**accessed values**		**I/O cost**	
Pattern	$value_j(ALT_{Oracle})$	$value_j(ALT_{ICE})$	$value_j(ALT_{Oracle})$	$value_j(ALT_{ICE})$
Data Access	*7,582,059.55*	*-13,900,442.5*	*3,695.56118*	*-6,775.1955*
Non-vector	*-132,898.09*	*97,458.6*	*-1,804.97727*	*1,323.6500*
Group By	*-570,150.00*	*627,165.0*	*-2,941.56914*	*3,235.7260*
Sort	*-16,862.55*	*4,637.2*	*-274.34436*	*75.4447*
Sum	*-108,192.73*	*119,012.0*	*60.79518*	*-66.8747*
Projection	*-297,692.73*	*196,477.2*	*-3,253.72114*	*2,147.4560*
$\sum value(ALT_i)$	*6,456,263*	*-12,855,692*	*-4,518.25600*	*-59.7935*
	Case C			
Workload	**accessed values**		**I/O cost**	
Pattern	$value_j(ALT_{Oracle})$	$value_j(ALT_{ICE})$	$value_j(ALT_{Oracle})$	$value_j(ALT_{ICE})$
Data Access	*2,780,088.50*	*2,780,088.50*	*1,355.0391*	*1,355.0391*
Non-vector	*-175,425.48*	*175,425.48*	*-2,382.5700*	*-2,382.5700*
Group By	*-752,598.00*	*752,598.00*	*-3,882.8713*	*-3,882.8713*
Sort	*-16,693.92*	*16,693.92*	*-271.6009*	*-271.6009*
Sum	*-428,443.20*	*428,443.20*	*240.7489*	*240.7489*
Projection	*-235,772.64*	*235,772.64*	*-2,576.9471*	*-2,576.9471*
$\sum value(ALT_i)$	*1,171,155*	*-1,171,155*	*-7,518.2010*	*7,518.2010*
	Case D			
Workload	**accessed values**		**I/O cost**	
Pattern	$value_j(ALT_{Oracle})$	$value_j(ALT_{ICE})$	$value_j(ALT_{Oracle})$	$value_j(ALT_{ICE})$
Data Access	*1,390,044.2*	*-1,390,044.2*	*1,129.1992*	*1,129.1992*
Non-vector	*-219,281.9*	*219,281.9*	*-3,309.1250*	*3,309.1250*
Group By	*-940,747.5*	*940,747.5*	*-5,392.8767*	*5,392.8767*
Sort	*-20,867.4*	*20,867.4*	*-377.2235*	*377.2235*
Sum	*-535,554.0*	*535,554.0*	*111.4578*	*-111.4578*
Projection	*-65,492.4*	*65,492.4*	*-1,193.0311*	*1,193.0311*
$\sum value(ALT_i)$	*-391,898.9*	*391,898.9*	*-9,031,5990*	*9,031,5990*

Table 5.12.: *Case B-D: Partial results for value(ALT_i) per pattern and cost function and overall value(ALT_i) per system and cost function.*

ing itself has an uniformly distributed higher impact[20] (i.e., $weight_j$ is $9/50$ with $\{j|j \in T \backslash Data\ Access\}$). One may argue, data access can be highly parallelized (e.g., with RAID or SAN solutions) in some use cases and its more important that data size drops fast after data access and evaluation of selection criteria due to limited shared memory. We observe fast drop in data size after data access for ICE[21] (cf. Tables 5.6 (on Page 86) and 5.9). We recognize for ICE that over 70% of estimated cost is imputable to the data-access pattern. That is, we observe in the exemplary workload on ICE approximately 94% of cost concerning accessed values and we observe approximately 76% of cost concerning I/O, respectively. Furthermore, we observe in the shifted workload approximately 90% of cost and approximately 72% of cost, respectively. Consequently, we determine a fast drop in data size and a low (relative) share of cost for the other patterns. That is, ICE evaluates large data sets in the beginning; but in comparison thereto, ICE only processes small data sets.

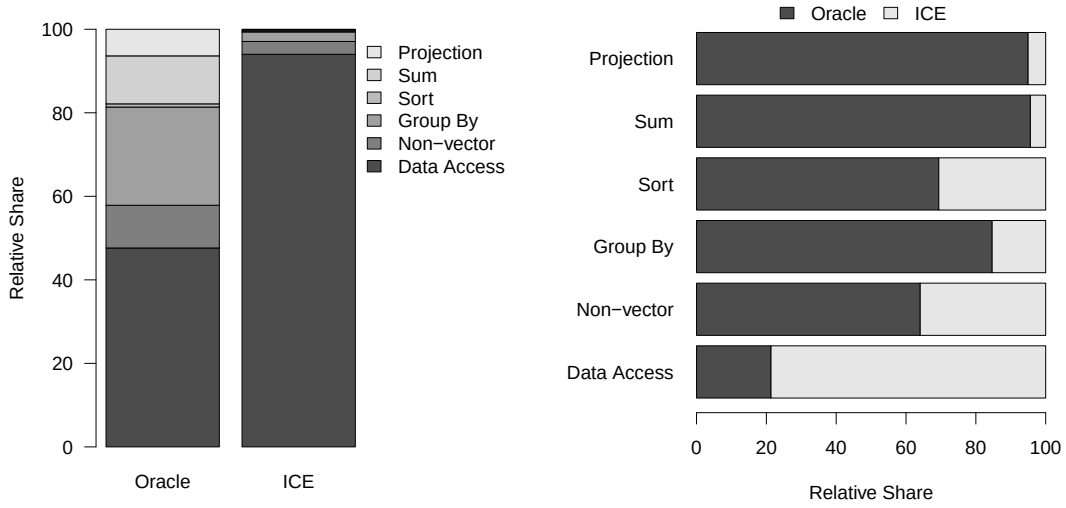
We visualize this behavior for the exemplary workload in Figures 5.8(a) and 5.9(a). However, we apply $weights_j$ to the according $value(ALT_i)$ and present the results for Case C in Table 5.12. Again, we observe no change in ranking between $value(ALT_i)$ neither per cost function nor between cost functions themselves. That is, we observe higher $value(ALT_{Oracle})$ concerning accessed values (i.e., $1{,}171{,}155 > -1{,}171{,}155$) and lower $value(ALT_{Oracle})$ concerning I/O (i.e., $-7518.201 < 7518.201$) –for ICE it is vice versa. Furthermore, we summarize the relative share of accessed values per DBMS for each workload pattern in Figure 5.8(a). In Figure 5.8(b), we summarize the relative share of DBMSs for each workload pattern to show the impact of different processing schemes to costs. In Figures 5.9(a) and 5.9(b), we illustrate the corresponding values for I/O cost.

For Case D, we develop our idea further that we are only interested in processing of data rather than access (i.e., data-access pattern) and presentation of data (i.e., projection pattern). Hence, we reduce $weight_j$ for data access and projection (i.e., $weight_j$ is $1/20$ with $\{j|j \in \{Data\ Access, Projection\}\}$); whereas we increase $weight_j$ for other patterns (i.e., $weight_j$ is $9/40$ subject to $\{j|j \in T \backslash \{Data\ Access, Projection\}\}$).

We observe a change in ranking between $value(ALT_i)$ for Oracle and ICE in Case D. That is, $value(ALT_{ICE})$ is higher than $value(ALT_{Oracle})$ for both cost functions now (i.e., $\sum value(ALT_{ICE}) > \sum value(ALT_{Oracle})$ for accessed values and I/O cost; cf. Table 5.12). Hence, we observe higher $value(ALT_{ICE})$ concerning accessed values (i.e., $391898.9 > -391898.9$) now just as well as we observe higher $value(ALT_{ICE})$ concerning I/O cost again (i.e., amount of $value(ALT_i)$ changes as expected; $9031.599 > -9031.599$). Consequently, we determine that weights have an impact on the design decision – at least for Case D. Therefore, we show four passes of our offline benchmarking model (i.e., four pairwise rankings) with four different (to some extent artificial) weighting functions (cf. Tables 5.10, 5.11, and 5.12). We

[20] For example, data is (mostly) stored in main memory.

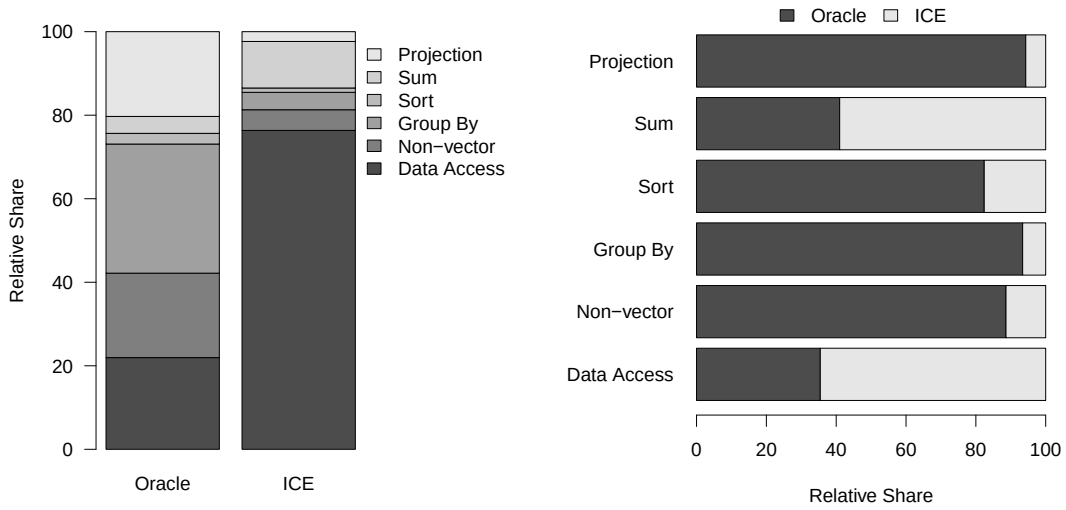
[21] We note, data access is (always) worst-case estimation in ICE. That is, data to be evaluated is estimated and not data to be accessed.



(a) Relative cost share of workload pattern for Oracle and ICE based on our exemplary workload.

(b) Cost share (Oracle vs. ICE) per workload pattern.

Figure 5.8.: *Comparison of accessed values for Oracle and ICE (cf. Table 5.6).*



(a) Relative cost share of workload pattern for Oracle and ICE based on our exemplary workload.

(b) Cost share (Oracle vs. ICE) per workload pattern.

Figure 5.9.: *Comparison of I/O cost for Oracle and ICE (cf. Table 5.6).*

note, the weighting functionality in our (offline) benchmarking model is applicable to DBMS-provided statistics as well as for predicted workloads. Furthermore, we argue that our benchmarking approach can be on arbitrary degree of detail (like cost representation; cf. Section 4), thus, we are still able to compute the design decision on arbitrary degree of detail assuming that the degrees of freedom are compatible.

We conclude, our approach is capable to advice architectural design with "weighting" functionality. On the one hand, we present query-frequency representation in our design-prediction model (cf. Section 5.2.2). Therefore, we map the frequency of a query – which consists of a number of tasks j – to the probability function $p(i,j)$, thus, $p(i,j)$ represents the frequency distribution of tasks j without uncertainty in a workload (cf. Equation 5.7).

On the other hand, we present the integration of user preferences (i.e., $weight_j$) with our (offline) benchmarking model (cf. Section 5.2.3). We show four cases (Case A-D) for $value(ALT_i)$ computation – which we derive from $u(P)$ according to Fishburn [Fis67a, Fis67b] – and its impact on the design decision. For Case B and C, we do not observe a crucial impact in comparison with the unweighted computation (Case A) (i.e., each system is in favor of one cost function concerning $value(ALT_i)$). For Case D, we observe an impact on the design decision due to the fact that $value(ALT_i)$ is in favor of both cost functions for one system now. Thereby, $weight_j$ represent the frequency of tasks for Case B; whereas $weight_j$ are artificial for Case C and D.

5.4.2. Design heuristics

In this section, we show another outcome from our work that is partially related to the weighting functionality and to the user preferences. That is, we present heuristics – a guideline – according to workload types and resource consumption in the following. We argue, heuristics give an idea of how to select a suitable architecture on restricted information or how to improve the architecture-design decision by means of user preferences (e.g., information on bottlenecks on certain resource).

As our DM is not restricted to one architecture, we do not restrict ourselves considerations to a certain architecture. Furthermore, we consider a set of heuristics that give an outline which architecture is suitable for an application (field). Some valid rules exist. One rule can be, pure OLTP applications perform best on Row Stores. A second rule is, (classic) OLAP application with an ETL process or rare updates are efficient on Column Stores [22]. However, we consider the gray area between both extrema (i.e., between pure OLTP and classical OLAP). In the following, we dedicate ourselves to the question: In which situation does one architecture outperform the other one and when do they perform nearly equivalent?

OLTP. For OLTP workloads, we just can recommend using Row Stores for efficient ACID-support on frequent DML statements. Tuple reconstructions on updates and

[22] Note: Not all Column Stores support update processing; they support just ETL (e.g., ICE).

data partitioning on inserts decrease significantly the performance of Column Stores in this domain. A Column Store does not achieve competitive performance except column-store architecture changes significantly to support concurrent update processing (with ACID-support) efficiently (e.g., cf. Abadi et al [AMH08]).

OLAP. In the OLAP domain, one might assume a similar situation as for OLTP workloads; whereas we argue that this is not true in general. We are aware that Column Stores outperform Row Stores for many applications or queries; that is, for aggregates and data access as well as for processing of a small number of columns, Column Stores perform better. In the majority of cases, Column Stores are most suitable for applications in this domain. Nevertheless, we state that there are complex OLAP queries where Column Stores ' advantage drops (cf. Section 3.2). Complex OLAP queries are mostly composed of consecutive groupings, a large number of predicate selections on different columns, or complex joins. For such queries, Row Stores show competitive results although they consume more resources (e.g., main memory, disk space). We argue, the high amount of tuple reconstruction within these queries drops Column Stores' performance. Consequently, these queries have to be considered for an architecture selection in more detail because they have a critical impact on physical-design estimation (e.g., by our DM; cf. Section 5.2). This applies to all workloads that contain more than a few queries of this class.

OLTP/OLAP. For mixed scenarios, the physical design strongly depends on the ratio between updates, point queries, and analytical queries. We assume that OLTP and OLAP workload have to be processed on same most up-to-date data (but no claim to real-time processing). Our experience is that Column Stores perform about 100-times slower on OLTP transactions (e.g., updates) than Row Stores. That is, Column Stores have to reconstruct tuples for update processing[23] to identify those column values that have to be updated; whereas Row Store identify necessary tuples and compute directly on these. In practice, this observation is more crucial because it does not even consider concurrency; that is, our observation bases on single-user execution of transactions. Assuming transaction and analytical queries take the same time in mean, we state that a transaction (OLTP) shall only occurs every 100 queries (OLAP) on Column Stores. Moreover, we argue that analytical queries last longer than single transactions. That is, we determine a smaller ratio. Our experience is, ten (executions of) analytical queries create higher advantage on a Column Store than extra costs caused by a single transaction. Furthermore, we may not give a distinct advice whenever the ratio is smaller than 10 : 1 (i.e., analytical queries : transactions). We recommend using Row Stores for such scenarios; except that beforehand, one may know, the ratio is changing to the advantage of analytical queries. If the ratio is not only temporary below 10 : 1 for a Column Store then an architecture change may be appropriate. We conclude, in mixed workloads (OLTP/OLAP), the architecture decision is all about the ratio between analytical

[23] We exclude blind write without selection criteria.

queries and transactions. We note further, ratios queries versus transactions (i.e., 100 : 1 and 10 : 1) may change depending on (complexity of) OLAP queries. We discuss this issue in Section 6.2 in detail.

CPU & I/O. For physical design, we state that CPU and I/O load have to be considered. We observe that in average, Row Stores consume more I/O bandwidth as well as in peak bandwidth, because data size is larger a priori. We distinguish between estimation of to be evaluated data (cf. Section 5.3) and actual measurable I/O consumption (cf. Section 6.4). We observe that the measurable I/O consumption is far less than the I/O which has been estimated[24]. Due to tuple reconstruction and decompression of data, Column Stores consume more CPU time because reconstruction and decompression are additional computational cost. That is, we have to consider wherever we have reserves in hardware resources (e.g., for load balancing). For more detailed discussions on CPU and I/O consumption, we refer to Chapter 6.

Our heuristics can guide the direction for architecture decisions for certain applications. That is, we may select a suitable architecture for an application and afterwards use existing approaches (e.g., IBM's advisor [ZRL+04]) to tune physical design of the selected architecture. Whenever workload and/or DBSs are available for analysis, we recommend the usage of our DM [LKS11a] to compute the optimal architecture for a workload. The above described heuristics for physical design extend our DM to reduce computation cost (i.e., solution space is pruned). Additionally, heuristics make our DM available for scenarios where no or only less information is available.

5.5. Summary

We defined and discussed cost estimations and introduce our storage-advisor approach in Chapter 5 that allows us the computation of architecture-design decisions. We used the example of Column Store and Row Stores to explain our approach.

We combined our storage-advisor approach with the workload-pattern approach to represent the necessary information for decision making. Therefore, we discussed challenges for cost functions and their parameters (cf. Section 5.1), which are necessary for computation of an optimization problem. An optimal storage architecture has to satisfy all or at least a weighted optimum of all optimization criteria. We argued, complex (composite) cost functions for several cost measure are not feasible to advise the optimal storage architecture because advantages as well as disadvantages are hidden in these functions. Advantages and/or disadvantages emerge due to different processing schemes of architectures. However, we argued that a weighted approach for several criteria is feasible; whereas we took the ranking of alternatives according to given criteria into account. For ranking, we introduced cost function $C(i,j)$ that maps a certain cost measure to the corresponding cost criterion. We

[24] One may argue, optimizers for Column Stores do not have yet the maturity as Row Stores have which have been developed for decades.

discussed different cost measures and the corresponding cost functions which are usefull for ranking of alternatives. Furthermore, we argued that we compute the storage-architecture selection on arbitrary degree of detail and thus, on arbitrary cost functions. Hence, the degree of detail for our computation is only dependent on detail degree of cost measures (i.e., stored information in workload patterns) that can be arbitrarily aggregated (cf. Section 4.2). We concluded that our approach is transparent to cost functions.

We developed our storage advisor in terms of Decision Models (DMs). Our idea goes back to the Knapsack which is a common approach for the enumeration in (database) physical design (i.e., we selected subset of candidates to maximize profit). We introduced our derivation of the (0-1) Knapsack to an abstract DM. We argued, we had to take into account multiple architectures to compare these to each other. Therefore, we extended our abstract DM to a multiple-choice Knapsack. That is, we introduced a new variable i that represents the architecture, whereas we select candidates from i classes with $i \in \{CS; RS; \ldots; m\}$ now. In the following, we derived three DMs from the abstract model.

First, we introduced our *online DM* that uses DBMS-extracted statistics (cf. Section 5.2.1). That is, statistics are available without uncertainty[25], and thus, we select the storage architecture based on cost estimates. We argued that we had to adapt our abstract model due to inversion of optimization criterion. Therefore, we computed the storage-architecture selection with costs without resource constraint instead of profit computation according to a given capacity constraint. We did equate our DM with the classic assignment problem.

Second, we introduced our (offline) *design-prediction model* that provides storage-architecture decision whenever DBMS statistics are not available (cf. Section 5.2.2). That is, we computed with predicted workloads the storage-architecture decision. Therefore, we had to consider uncertainty of cost estimation. We argued, our approach is independent from query structures, but it is not independent from query cost structures (cf. Chapter 3 and Section 4.3). Moreover, we had to consider the uncertainty of a Multi-Criteria Decision Problem (MCDP) due to several tasks within queries which generate multi-dimensionality. We extended our cost function ($C(i,j)$) in that manner that we combined uncertainty of cost estimation with computation of cost. That is, we introduced cost function $C^*(i,j,)$ that is composed of $C(i,j)$ and a probability function $p(i,j)$. The probability function $p(i,j)$ represents the probability that a task has to be computed in a workload; whereby the sum of probabilities $p(i,j)$ has to be 1. We combined our consideration to our (offline) design-prediction model that was derived from our online DM.

Third, we introduced our *offline benchmarking model* that combines ideas of the online DM and the design-prediction model. That is, we combined the capability to benchmark two (or more) database systems with the capability to compute on samples. Therefore, we had to solve a MCDP (under uncertainty) due to the fact that we have no access to environmental parameters and desired system behavior. In

[25]Moreover, statistics are stored in our workload patterns normalized (cf. Chapter 4).

the MCDP, we represented the environmental parameters via user preferences. We had to qualify two aspects to classify our MCDP: (a) is an order of alternatives is available and (b) what is the type of order. Therefore, we considered Multi-Criteria Decision Analysis (MCDA) methods with functional preference due to the fact that we computed decisions based on costs which supposed to be in algebraic order. We used Multi-Attribute Utility Theory (MAUT) whenever costs values and a utility function were available. Hence, we had to consider if the type of order is applicable to a utility function. That is, we had to determine if an interval measurement with arbitrary-defined zero point is given. Due to the fact, we computed the storage-architecture decision with utility values (i.e., difference of positive alternatives' cost), we fulfilled the conditions for utility functions. Consequently, we integrated our decision-making methodology in the general MAUT method. We showed that we are able to compute a ranking for a set of alternative storage architectures due to the MAUT methodology with respect to user preferences. However, we argued that our DMs – regardless of the extensions – are transparent to cost functions and they still support storage-architecture decision on arbitrary degree of detail.

We evaluated our online DM in Section 5.3 to show soundness of our approach. Therefore, we processed our exemplary workload composed of queries `Q6`, `Q15`, and `Q16` from the TPC-H benchmark on a Column Store as well as on a Row Store. We presented the statistic-gathering process based on our exemplary workload (cf. Section 5.3.1). We observed divided results for our exemplary workload concerning data access; that is, the Row Store read fewer rows for `Q6`, `Q15`, and `Q16`. We argued, the Column Store scanned several columns on large relations (e.g., fact table `LINEITEM`) for `Q6` and `Q15` completely; whereas the Row Store read only a smaller amount of tuples. Furthermore, we observed that I/O may be directly correlated to accessed rows. We observed higher I/O cost for the Column Store concerning `Q6` and `Q15`. In contrast, we observed surprisingly higher I/O cost for the Row Store concerning `Q16`. We contrasted rows with (column) values in previous considerations. Hence, we applied our statistic normalization approach to calculate accessed (column) values for Oracle. We observed that the Row Stores read still fewer values for `Q6` and `Q15`, but for `Q16` the Column Store read fewer values now. We stated that the correlation between accessed rows and values as well as I/O holds only if statistics are normalized. We considered two additional queries from the TPC-H benchmark (i.e., `Q4` and `Q17`) to show that the observed behavior especially for `Q16` is not an artifact. In the following, we showed our solution process for the assignment problem that we had to solve in online DM. Therefore, we built up a linear program that we implemented in A Mathematical Programming Language (AMPL)[26]. We figured out that as a result, our decision mode computed a draw between the Row Store and the Column Store. That is, the Row Store computed approximately half number of (column) values then the Column Store did (i.e., ratio is approximately $1 : 2$); whereas the Row Store caused twice as much I/O cost as the Column Store for computation (i.e., ratio is approximately $2 : 1$). Finally, we discussed potential improvements to

[26]For details, we refer to Fourer et al. [FGK02].

the DM as well as ideas to improve results (e.g., soundness).

In Section 5.4, we showed improvements to the decision-making process. We discussed the necessity of query-frequency representation as well as the mapping of user preferences (cf. Section 5.4.1). Therefore, we showed the impact of query frequency to the decision-making process with a shifted workload. The shifted workload differs from our exemplary workload in this manner that query `Q16` is executed twice. We used our design-prediction model (cf. Section 5.2.2) to compute storage-architecture decision according to different query frequencies. Therefore, we mapped query frequency to frequency of corresponding tasks with respect to the total number of tasks in a workload. That is, the probability function $p(i,j)$ represents the frequency of tasks now; and thus, it represents the query frequency without uncertainty. In summary, we observed in the shifted workload that the ratio between Oracle and ICE for number of accessed values remained (approximately $1:2$); whereas the ratio for I/O cost shifted from approximately $2:1$ to approximately $3:1$. We argued that the evaluation slightly changed on behalf of ICE. Furthermore, we showed the integration of user preferences (i.e., weights) by example. We used our (offline) benchmarking model to compute the storage-architecture decision with weights (cf. Section 5.2.3). Therefore, we showed the computation for four cases (i.e., Case A to D); whereas Case A corresponded to the unweighted computation to show the model switch had no impact on the storage-architecture decision. For Case B, we used the frequency of tasks as weights that did not change the ranking of alternatives; that is, the amount of utility values changed but not the ratio of utility values between Oracle and ICE. We used artificial weights for Case C and D to show the (possible) impact of user preferences to the storage-architecture decision. We argued in Case C that the data access did not have the same impact as the processing of data itself. Therefore, we reduced the weight for the data-access pattern; whereas the other pattern weights were increased. We figured out that the amount of utility values changed but not the ratio between Oracle and ICE again. For Case D, we developed further the idea that data processing in contrast to access and presentation of data has a serious impact on the storage-architecture decision. Therefore, we reduced further the weight for data access as well as the weight for data projection was reduced. The weights of the other patterns were increased accordingly. We figured out that the amount of utility values changed as well as the ratio of utility values between Oracle and ICE. That is, we observed that both cost measures were on behalf of ICE now; in contrast to one cost measure on behalf of each system for Case A to C.

6. Hybrid Storage & Query Processing

Chapter 6 shares material with [LS10, LKS12, WKLS12, LSS13, LSKS14].

We introduced our approaches to store workload statistics in Chapter 4. Furthermore, we presented our storage advisor in Chapter 5, which recommends the optimal storage architecture for (re-) design decisions. In this chapter, we present a holistic approach that analyzes queries and recommends an architecture from a given set for query execution independent from the underlying environment. Therefore, we use approaches from previous chapters for query dispatching and query optimization in hybrid storage-architecture environments, respectively. Due to transparency from queries' point of view, we denote our approach as query interface.

The query interface comprises the complete decision process (cf. Figure 6.1). That is, we recommend an architecture with our Decision Models (DMs) that compute on gathered statistics. We reduce the existing workload analysis to a query analysis with respect to where we execute a certain query. Therefore, we use so-called what-if analysis [CN98, ZZL+04] and heuristics to give recommendation for a corresponding query. For the what-if analysis, we use our online DM (cf. Section 5.2.1) on hybrid systems; whereas hybrid systems maintain both stores[1] redundantly. We distinguish between Hybrid Database Management System (HDBMS) (i.e., both stores in one DBMS) and Hybrid Database System (HDBS) (i.e., both stores in different DBMSs). We denote the above described approach as (online) query-dispatching module. Together with our (offline) storage-advisor module, the query interface adds up to the Automated Query Interface for Relational Architectures (AQUA^2) framework; whereas the heuristic framework is implemented crosscutting in AQUA^2's offline and online part. Despite design recommendation via storage advisor, the design-prediction model and the offline benchmarking model act as design advisor or design alerter within HDBMS or HDBS without replicated stores (e.g., applying profit and modification costs; cf. Sections 5.2.2 and 5.2.3). AQUA^2 implements our query interface for relational architectures. Thereby, we argue that the interface is hybrid (a) due to merging of the storage-advisor and query-processing approaches with a crosscutting heuristic framework and (b) due to the transparent support of different architectures.

First of all, we introduce general ideas for our hybrid query interface in Section 6.1. In Section 6.2, we discuss heuristics for hybrid DBS and DBMS; whereas we discuss data locality and degree of freshness in Section 6.3. We evaluate our online dispatcher approach in Section 6.4.

[1] We refer to column and row store.

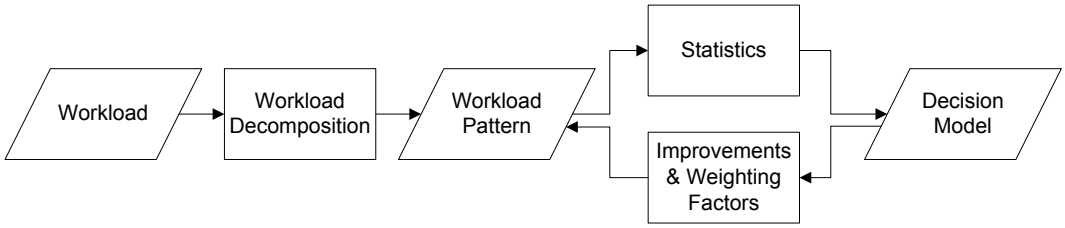


Figure 6.1.: *Decision process comprised by the hybrid query interface.*

6.1. A Hybrid Query Interface

We introduce our query interface for hybrid stores in this section. We present a holistic overview of our AQUA^2 framework [LSKS14]. First, we discuss the integration of our design advisor (cf. Chapter 5), and then we introduce rule-based query optimization (i.e., dispatching of queries). Therefore, we cover the complete process from workload (respectively query) to decision and dispatching. Second, we discuss feasible optimization levels for query optimization in HDBMS/HDBS.

6.1.1. Introduction of AQUA^2

The combination of the OLTP and OLAP domains in one application scenario raises the question of how to process mixed workloads. Therefore, we develop the AQUA^2 framework (cf. Figure 6.2) that considers design prediction and performance estimation across column- and row-oriented architectures[2]. AQUA^2 refers to our storage-design advisor (cf. Figure 6.2(a)) as well as to dispatching of queries in HDBMS/HDBS (cf. Figure 6.2(b)).

We integrate our design-advisor approach to select an optimal storage architecture for a given (sample) workload based on statistic-storage and -aggregation approach [LKS11c] (cf. Figure 6.2(a)). On the one hand, we are able to extract workload statistics from existing DBMSs (e.g., by explain-plan functionality). On the other hand, we can use estimated workload statistics (i.e., samples) for our storage advisor. We store the statistics using our workload-representation approach in both cases. Our DMs use these statistics to recommend the storage architecture [LKS11a]. We point out that the uncertainty of decision is dependent on source of workload statistics. However, there may be cases that statistics or sample workloads are not available. Therefore, we develop a set of heuristics for physical design (i.e., rule-based physical-design approach) to recommend the optimal storage architecture [LSKS12, LKS12]. The storage advisor and its extensions are summarized in Figure 6.2(a) as the ***Offline-Decision Framework*** of AQUA^2 (for more details cf. Chapters 4 and 5). We are only able recommend meaningful architectural decisions for those workloads that in some manner (at least slightly) are dominated by one domain (i.e., OLTP or OLAP). For other (mixed) workloads, we encourage devel-

[2]Please note, we are not limited to these two architectures in general (cf. Section 5.2).

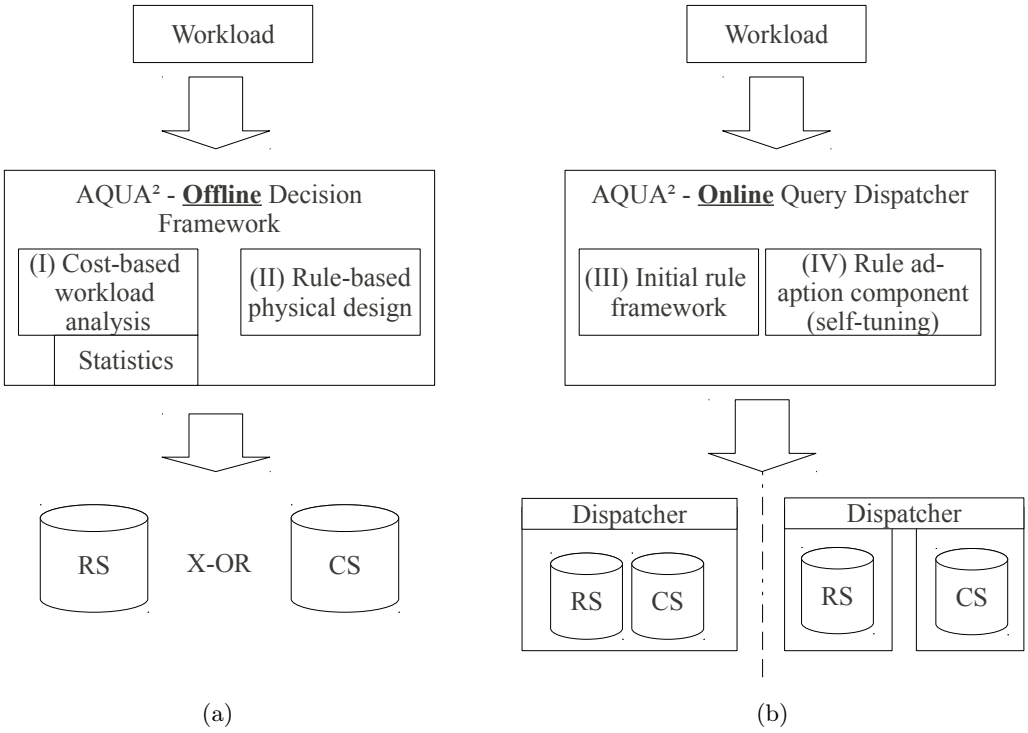


Figure 6.2.: *Overview of AQUA^2's core components: (a) the storage-advisor module and (b) the query-dispatching module.*

opment of hybrid stores. Moreover, we suggest that hybrid stores are more efficient than each architecture alone is.

To show the efficiency of hybrid stores, our query interface has to support hybrid stores – regardless it is an HDBMS or an HDBS – to encourage the development of such systems. Hence, we adapt our rule-based architecture-selection approach to query-dispatching approach for mixed workloads (cf. Figure 6.2(b)). We consider hybrid stores with replicated data for now. That is, we use two types of hybrid stores where data is available (i.e., replicated) in both stores. First, we investigate HDBSs that comprise two DBMSs (i.e., one column and one row store). We assume replicated data across both stores, and we use the dispatcher functionality to distribute queries according to our rule set. We denote this case as the *replicated solution*. Second, we consider HDBMSs that comprise column-oriented as well as row-oriented storage in one DBMS. We assume that HDBMSs maintain data redundantly in column- and row-oriented storage; whereas queries are dispatched to the appropriate storage variant internally. This case is the *hybrid solution*. Our interface – AQUA^2 – supports both hybrid-store types.

We claim that the rule-based query dispatcher is the most promising approach for query optimization on hybrid stores, because in terms of complexity cost-based query dispatching is similar to cost-based query optimization for *two* stores though.

In consequence, cost-based optimization for dispatching (in the interface) and for query execution (in the store) is very costly. We discuss different optimization levels and strategies in Section 6.1.2. Therefore, we extend our rule-based physical-design approach to a rule-based query dispatcher. We extend our set of heuristics with query-processing heuristics [LSKS12, LKS12] for hybrid stores (i.e., replicated and redundant; cf. Section 6.2). That is, we analyze single queries – instead of workloads – and decide where to execute these best according to our set of rules. Whenever we determine where a query should be executed we dispatch this query to the according store (i.e., to the corresponding DBMS (HDBS) or query processor (HDBMS)). We argue, our approach is suitable for inter- and intra-query parallelization. That is, we dispatch queries according to current system load respectively to maximize the throughput for inter-query parallelization which is suitable for HDBMS and HDBS. Or otherwise, we dispatch sub-queries (i.e., part of a query) to different stores for intra-query parallelization, which can be beneficial for complex queries with multiple processing schemes. However, we argue that intra-query parallelization is more suitable for HDBMS due to simpleness of intermediate-result exchange. Nevertheless, we state that we would need a separate query engine for intra-query parallelization; whereas such (global) query engine contradicts the idea of our query interface.

We propose further a self-tuning component that adapts ideas of self-tuning [CN07] (e.g., index configurations or views). We argue, (self-) tuning (e.g., enumeration problem, parameterization) is crucial for the overall performance of DBS [Lüb09, SGS+13]. To the best of our knowledge, no current DBMS supports both architectures and is able to self-adapt the storage system[3]. Therefore, we implement a prototypical HDBMS to evaluate our approach for both hybrid-store types. Our prototype supports both architectures (i.e., redundant storage) to make the first step and gradually improve our solution (cf. Section 6.4). That is, we implement a rule-based query-dispatching approach in our prototype. We denote the discussed components of AQUA^2 as the ***Online Query Dispatcher*** (cf. Figure 6.2(b)).

6.1.2. Global vs. Local Optimization

For each DBMS, query optimization is a basic task. Cost-based optimizers are commonly used in DBMS to achieve optimal performance. Nevertheless, we argue that cost-based optimization is very costly due to computational complexity [SAC+79, Ioa96, KPP04]. Therefore, rule-based approaches are introduced to prune the solution space for cost-based optimization [Fre87, Sel88]. To the best of our knowledge, neither rule-based nor cost-based optimizer exists for our requirements. For hybrid stores, we have to consider different level of query optimization due to the fact that queries are analyzed in different steps. We analyze queries for execution on the optimal store in our query interface as well as queries are analyzed for query execution by the store to be dispatched. That is, we have to consider global and local optimization analog to distributed DBMSs [ED95, ÖV11].

[3] We refer to our analysis in Chapter 7.

In HDBMSs/HDBSs, we have to consider cost-based and rule-based optimization for local and global optimization levels. First, we perform optimization – regardless if rule- or cost-based – on the global level; that is, we need an architecture-independent optimization approach. This is the main task of AQUA^2. And second, we argue that local optimization is dependent on the corresponding architecture. Hence, we propose the application of existing optimization approaches (i.e., query engines with optimizer) of Row Stores and Column Stores for the architecture-dependent optimization – the local level. We present a rule-based optimization approach for our interface due to the lack of architecture-independent (cost-based) optimizer. We optimize queries on the global level based on architecture-independent heuristics and rules (cf. Section 6.2); what in our case corresponds to query dispatching without query rewriting. Queries are locally rewritten by architecture-dependent optimizers on either architecture. Consequently, we reduce the solution space from global to local optimization with our interface. At local optimization level, we reuse existing functionality. Rule- and/or cost-based optimization is common in DBMSs. Hence, we state that native optimizers (local optimization) achieve best performance (i.e., optimization result) because they are tailor-made implemented for the corresponding DBMS and thus, for the corresponding architecture. Furthermore, we cause minimum possible overhead for the overall optimization with our approach, because query optimization remains unchanged and rule-based query dispatching only adds slight computational overhead.

Finally, an approach for global cost-based optimization is also conceivable. Such an approach can be promising for global optimization goals (e.g., minimal total time). However, we argue that the solution space significantly increases for cost-based optimization, whenever we apply it to two architectures. That is, the computational cost also increases for cost-based query plans. As cost-based optimization causes high computational cost for single architecture already, most (commercial) DBMSs first prune the solution space by rule-based optimization [GD87, Fre87, Sel88]. We suggest that an architecture-independent optimizer for several architectures does not achieve competitive results compared to a tailor-made optimizer for a certain architecture and system. Consequently, we implement rule-based optimization in our query interface; whereas we keep cost-based query optimization at the corresponding architecture (i.e., its query optimizer).

6.2. Heuristics on Hybrid DBS and DBMS

We emphasize the usage of rule-based optimization on global level due to computational complexity for cost-based optimization and advantages of tailor-made optimizer on local level. Therefore, we present heuristics – an additional outcome from our work – for query execution on HDBSs and HDBMSs[4], respectively. We remind to our discussion on architectural design in Section 5.4.2, whose results partly overlap

[4]That is, we assume that the system supports column- and row-store functionality with replicated respectively redundant data.

```
1 SELECT SUM(l_extendedprice * l_discount) AS revenue
2 FROM lineitem
3 WHERE l_shipdate >= date '1994-01-01' AND l_shipdate < date '1994-01-01' + interval '1' year
    AND l_discount BETWEEN .06 - 0.01 and .06 + 0.01 AND l_quantity < 24;
```

Figure 6.3.: *TPC-H query Q6 [Tra10].*

with results in this section. We discuss only query-processing heuristics for OLAP and mixed workloads due to our observations that column stores are not competitive to row stores for OLTP in general.

OLAP. In the OLAP domain, we face a huge amount of data that in general is not frequently updated. Column stores reduce the amount of data significantly due to aggressive compression. That is, more data can be loaded in main memory as well as overall I/O between storage and main memory is reduced. We state, I/O reduction is a major benefit of column stores. We observe that row stores perform on many OLAP queries worse due to fact that CPUs idle most time while waiting for I/O from storage (cf. Section 6.4). Moreover, row stores read unnecessary data for most OLAP queries (e.g., query `Q6` (cf. Listing 6.3) from TPC-H benchmark [Tra10]). That is, we observe that only a few columns of schema's relations are accessed (e.g., 4 out of 16 for `Q6`). Simultaneously, aggregate functions belong to this execution schema, too (i.e., usually aggregates refer to one column). We state, column stores outperform row stores for OLAP as long as only a minority of columns has to be accessed for aggregation and predicate selection. We only dispatch OLAP queries to row stores for load-balancing reasons.

Complex OLAP Queries. OLAP analysis becomes more and more complex; thus, queries become more complex, too. Complex OLAP queries describe complex issues and generate large (complex) reports (e.g., query `Q13` may be part of a report or of a more complex query). For complex OLAP queries, a major part of relations have to be accessed and aggregated (i.e., nearly the whole relation has to be read). We argue, such queries imply a number of tuple reconstructions that reduce performance of column stores significantly. Hence, row stores achieve competitive performance for these queries. For `Q13` (cf. Listing 6.4), we observe another issue that causes a number of tuple reconstructions – group operations; that is, tuples have to be reconstructed precociously. We state that selection on dependent predicates, complex joins, and sub-queries are further issues that reduce performance through tuple reconstructions significantly. We argue, these queries are candidates to be dispatched to row stores (e.g., to balance load).

OLTP/OLAP. For mixed workload environments, our first recommendation is to split workload into two parts – OLTP and OLAP. We argue that both parts are allocated to the corresponding part of the HDBS or HDBMS (i.e., OLTP to row

```
1 SELECT c_count, COUNT(*) AS custdist
2 FROM (SELECT c_custkey, COUNT(o_orderkey)
3     FROM customer LEFT OUTER JOIN orders ON c_custkey = o_custkey AND o_comment NOT
        LIKE '%special%request%'
4     GROUP BY c_custkey) AS c_orders (c_custkey, c_count)
5 GROUP BY c_count ORDER BY custdist desc, c_count desc;
```

Figure 6.4.: *TPC-H query Q13 [Tra10].*

Feature	Column Store Behavior	Background
Space consumption	Reduced by factor ≈ 10 (HDD & main memory)	Aggressive compression (e.g., optimal compression per data type)
Compressed data	Query processing without decompression	Does not apply to all compression types nor to all operations
Data transfer	Reduced and less disk swapping	More data fits in memory due to aggressive compression
CPU consumption	Increased	Compression and tuple reconstruction cause CPU load
(Point) Lookup	Fast response	True for OLAP and OLTP workloads
Joins	Fast for foreign key join Slow for complex join (many columns) Slow for (full) outer join	Processing on indexes/single column Tuple reconstruction needed Tuple reconstruction needed
Predicate selection	Fast on independent predicates Slow on dependent predicates	May processed in parallel Highly dependent on structure and dependencies (e.g., reconstruct tuples)
Vector operations	Fast processing (e.g., bitmap join)	Inherently supported and easily adaptable
Aggregates and column operation	no I/O overhead	Occur frequently in OLAP
Parallelization	For inter- and intra-query (e.g. parallel aggregation)	Not for ACID transactions with write operations

Table 6.1.: *Insight summary of store qualities for design.*

store and OLAP to column store). With this approach, we achieve competitive performance for both OLAP and OLTP. We do not recommend a more fine-grained splitting for OLAP-workload parts due to the fact that intra-query parallelization demands for a separate query engine. Nevertheless, our split methodology can be extended for load balancing whenever one store is overloaded (i.e., we dispatch to other store when resources are available).

We summarize the major insights from above and from Section 5.4.2 in Table 6.1. We present our insights for column stores due to the fact that row store's behavior is contrary. Both architectures have advantages as well as disadvantages. We emphasize advantages of column stores (cf. Table 6.1); nevertheless, column stores still perform worse on update operations and concurrent non-read-only data access, respectively. That is, frequent updates and consistency checks cause tuple reconstructions on column-stores due to the inherently partitioned data; and thus, significant cost. In

Origin	Statement	Result
Transactions (DML)	Perform best on `RS`	Execute always on `RS`
(Point) Lookups	Perform well on `RS` and `CS`	Without overload on `RS` To load balance on `CS` if no real-time requirement
Aggregates	Perform best on `CS`	Execute always on `CS`
Grouping	Perform best on `CS` except for ∗-operator (e.g., TPC-H `Q4`) and ≥ 10 columns (e.g., TPC-H `Q1`)	Execute always on `CS` To `RS` for load balancing
Predicates	Distribute for parallelization only depending on type of predicate (e.g., TPC-H `Q8`, `Q13`)	Dependent predicates to `RS` Independent predicates to `CS`
Sub-query	Distribute for parallelization only dependent on sub-query type	To `CS`: Comprised of aggregates, independent predicates, foreign-key joins (e.g., TPC-H `Q2`) or to `RS`: for sub-queries with dependent predicates (e.g., TPC-H `Q11`)
Joins	Distribute for parallelization only depending on foreign-key, outer, multi-column join	Dispatch foreign-key joins to `CS` to get a key list Dispatch outer and multi-col. joins to `RS` for intermediates

Table 6.2.: *Rules for the online query dispatcher in AQUA²*.

analytical scenarios, row stores read a lot of unnecessary data because operations are often based on single columns or small subsets of them. Additionally, row stores do not reach compression ratios like column stores because different data types are combined in tuples that cannot be compressed on high ratio as columns that have just one data type and in best case a more uniform structure and length. That is, data size is already larger than for a column store which implies more I/O and more main memory consumption. In line with others [AMH08, ZNB08], we conclude that both architectures have the right to exist in different applications domains. Neither row stores cannot outperform column stores in their domain nor vice versa.

Therefore, we focus our approach on gray areas between domains (i.e., mixed (OLTP/OLAP) workloads, complex OLAP queries). That is, we introduce a heuristic-based decision framework (i.e., rules) that dispatches queries to the optimal store independently when we apply to HDBMS or HDBS [LSKS12, LKS12]. We assume for our set of rules that a consistency-control approach is available (i.e., replication mechanism). We present our rules for the online query dispatcher in Table 6.2. Hence, we present rules for inter- as well as intra-query parallelization as outcome of our research; even though intra-query parallelization demands for a (global) query engine (e.g., as for distributed DBMS [ED95, ÖV11]). However, our research is not focused on query-engine development; thus, our query interface supports currently inter-query parallelization. Consequently, we implement a subset of rules in a first step (cf. Section 6.4.3). We discuss improvements to our approach in Section 6.4.5 and refer to Section 6.3 for discussion on real-time and time-bound requirements. Nevertheless, we do not claim generality for design and query process-

ing on row-, column-, and hybrid stores. We are aware, further aspect for design and query processing for a domain exist, even though these may be no technical aspects. Some aspect are in favor of row stores:

Tuning: Self-tuning techniques and advisors are more advanced that simplify administration and tuning,

User behavior: Intuitive processing of SQL (i.e., easier to predict, understand, develop for),

Training: Most (IT-affine) people are familiar with row-store technology; thus, training costs are less,

Abstraction: Support for a wide field of data management tasks (i.e., mixed requirements with point- and range-queries, full scans for exploration).

6.3. Load Balancing & Queries with Time Constraints

We figure out that it is promising to dispatch mixed workloads (i.e., OLTP/OLAP) to different architectures [Lüb10, LS10, LKS11c]. Therefore, we derive our dispatching rules for the online query dispatcher (cf. Section 6.2). That is, as rule of thumb, we dispatch OLAP queries to the column stores and OLTP transactions to the row stores; whereas we define OLTP transactions roughly as DML statements and point lookups (e.g., cf. TPC-C benchmark[5]). Nevertheless, we observe promising results for load balancing, inter- and intra-query parallelization, and (near) real-time OLAP. For these optimization aspects, we argue that a decision where to best execute a query or transaction is insufficient. First, we need information of resource consumption and currently available resources (i.e., further optimization parameter) for load balancing and parallelization (e.g., current CPU and I/O load). The impact of hardware consumption is highly dependent on hardware setup; thus, we have to figure out which resource is most restricted to apply our rules correctly. Second, we need information of queries' requirements according to degree of freshness. That is, we have to know whether most up-to-date data is or is not necessary.

Besides query dispatching for optimal query execution, we can also use our approach for load-balancing in HDBSs and HDBMS with full data redundancy. That is, full data redundancy (in both column and row store) supports the execution of all OLAP queries and OLTP transactions on either architecture. On the one hand, we can compute cost-based query-plan comparison with our DM (cf. Section 5.2). That is, we dispatch a query to the store with least resource consumption with respect to a certain resource[6] (e.g., I/O bandwidth). We argue, optimization costs increase for this approach; and thus, computation time may be higher than execution time especially for transactions. On the other hand, we may dispatch queries rule-based

[5] `http://www.tpc.org/tpcc/default.asp`.

[6] We note, we are not restricted to one resource; however, we need a weighting methodology for several resources.

according to their domain. Therefore, load balancing promises an improvement of overall system throughput. That is, we implement inter-query parallelization with respect to the overall workload. However, we show real-time updates in column store are inefficient (cf. Section 6.4.2), thus, data synchronization slows down the overall query processing. We assume a similar behavior in an HDBMS. We validate this assumption with our prototypical implementation. Nevertheless, column stores show competitive performance on non-concurrent data access. Consequently, OLTP transactions only should be dispatched to column stores if these are comprised of lookups, which do not depend on most up-to-date data and do not involve concurrent access. Additionally, we propose load balancing for OLAP queries; especially queries that (nearly) perform equivalent on both system parts (cf. Sections 6.2 and 6.4.2). Hence, we dispatch OLAP queries to the row store if the column store is under high load. Consequently, our optimization criterion shifts from minimum query-execution cost (cf. Chapter 5) to optimal load factor for query dispatching.

We observe similar results for intra-query parallelization. As a result, we may only dispatch very specific parts of queries due to the fact that intermediate-result exchange, which causes extra computational cost, has to benefit in total. We number query parts to be dispatched among to row stores that cause high number of tuple reconstruction (e.g., groupings over all columns), (full) outer joins, joins with high number of join attributes, selection of dependent predicates, and sub-queries with dependent predicates. In contrast, we can dispatch foreign-key joins, aggregates, groupings, independent predicate selection, and sub-queries with previously named parts to column stores. However, our query interface does not support intra-query parallelization currently, as we mentioned before. Furthermore, we argue that intra-query parallelization is only useful for an HDBS if the HDBS is set up on a distributed environment and does not share hardware for column and row store. We state, we use DBMS monitoring, if available, to observe load peaks or alternatively OS tools. We emphasize that our approach may be enriched by further approaches (e.g. future-workload prediction [GCCK07]).

We further argue that time-bound queries cause another optimization criterion for our dispatching approach. We state, time-bound constraints occur due to time-bound data partitioning (i.e., data fragmentation [ÖV11, Pages 71 ff.]). In our environment (i.e., HDBS or HDBMS), we have an abstract horizontal partitioning due to the fact that data in the Column Store and data in the Row Store diverge from each other between two consistency-matching passes (i.e., ETL or replication mechanism; e.g., replica consistency with lazy master [PMS99]). That is, we have only most up-to-date data in the row store. This issue has an impact on OLAP processing – at least in terms of (near) real-time OLAP, because data has to be partially read from both stores or from the Row Store completely. We argue, there is no impact on standard reporting (e.g., for annual accounts for previous year). We state that the point in time, at which freshness of data becomes an issue, is dependent on the frequency of consistency-matching passes. Otherwise, there may be an issue for more up-to-date reporting (e.g., stock-receipts for current day).

We argue that our query-dispatching approach is suitable to support (near) real-

time OLAP queries. We propose two approaches to process (near) real-time OLAP queries. First, we compute queries, which demand for real-time, in the row-store part of hybrid HDBMSs or HDBSs; and thus, these queries are executed on most up-to-date data. Second, we use approaches from distributed DBMSs [ÖV11] to compute time-bound queries in a distributed way (e.g., freshness-aware middleware [RBSS02], adaptive virtual partitioning [LMV04, FLP+08]). For both approaches, we need (a) information from the application, whether a query demands for real-time data (i.e., a real-time flag), and (b) a separator (e.g., a timestamp) between current time and last consistency-matching pass. Under assumption that approaches for time-bound constraints and data locality are adapted and integrated, we process real-time queries on column store and on the row store in parallel. That is, we compute (normal) OLAP query on the column store, and for the time-bound constraint, we compute only most up-to-date data for OLAP queries on the row store. We may achieve better performance on time-bound queries with the (second) distributed approach that computes real-time queries like distributed DBMSs [ÖV11, Pages 205 ff.].

Nevertheless, the distributed approach for time-bound queries and intra-query parallelization in general demand for a (global) query engine that supports query processing on several architectures (including architecture-specific query decomposition), time-bound constraints, cost-based optimization, and resource monitoring. Our approach may be extended to support these demands, but currently we argue, a global query engine like in distributed DBMSs is not in our focus of current work for our query interface.

6.4. Evaluation – Online Query Dispatcher

In this section, we evaluate AQUA$^{\mathbf{2}}$'s *Online Query Dispatcher* to validate our claim that online dispatching of queries results in a better performance than an architecture decision. We point out that we switch the column store for evaluation due to the fact that Infobright ICE is a read-only DWH [Inf08, Inf11a]. In the following, we use Sybase IQ[7] 15.2 as column store that is able to process transactions and mixed workloads (i.e., OLAP and OLTP). That is, we use Sybase and Oracle to evaluate our approach for the *replicated solution*. Furthermore, we use our prototype for the *hybrid solution* (i.e., an HDBMS). Finally, we discuss extensions to our approach for hybrid stores (e.g., to reduce redundancy) in Section 6.4.5.

6.4.1. Evaluation Settings – Replicated Solution

In the following experiments, we use the standardized benchmarks TPC-H [Tra10] and TPC-C [Fer06][8] to show the significance of our results. We use scale factor 10 for TPC-H and 90 warehouses for TPC-C (i.e., both with approximately 10 GB of data). Additionally, we use the TPC-CH benchmark [CFG+11] – again with 90

[7] `http://go.sap.com/product/data-mgmt/sybase-iq-big-data-management.html`.
[8] A prepared TPC-C environment; Referring: `http://www.tpc.org/tpcc/default.asp`.

Query	Oracle	Sybase	Query	Oracle	Sybase
Q1	01:40	01:18	Q12	01:41	01:26
Q2	01:21	00:18	**Q13**	**00:52**	**00:42**
Q3	01:52	00:46	**Q14**	**01:24**	**01:16**
Q4	01:38	00:07	**Q15**	**01:22**	**00:19**
Q5	**03:03**	**00:22**	Q16	00:09	00:08
Q6	**01:21**	**00:03**	Q17	01:22	01:16
Q7	02:12	00:05	Q18	03:51	01:03
Q8	01:47	00:21	**Q19**	**01:23**	**02:03**
Q9	03:42	02:21	Q20	01:33	00:50
Q10	**02:00**	**00:15**	Q21	04:08	02:23
Q11	**00:13**	**00:09**	Q22	00:20	00:07

Table 6.3.: *Execution times of TPC-H queries (in mm:ss).*

warehouses – that simulates a mixed OLTP/OLAP workload. For the experiments, we use a Dell Optiplex 980[9]; whereby we measure CPU consumption and used I/O bandwidth for TPC-H and TPC-CH every 0.25 seconds, and for TPC-C every 0.01 seconds.

6.4.2. Evaluation – Replicated Solution

In the following, we evaluate the replicated solution for hybrid workloads (cf. Section 6.1.1). We use a column and a row store that keep data fully redundant and use a dispatcher to distribute the query workload. For our test scenario, we use an Oracle 11gR2.1 (row store) and a Sybase IQ 15.2 (column store) with each 1 GB main memory available. We present our results for the TPC-H benchmark in the beginning and followed by the results for the TPC-C benchmark. Subsequently, we present our results for the TPC-CH benchmark (i.e., a mixed workload).

First, we present the execution times for TPC-H on both systems in Table 6.3. We highlight queries in Table 6.3 that we discuss in more detail. We consider complex as well as classical OLAP queries. We identify easily without additional consideration that the row store cannot compete with the column store for `Q6`, but for `Q13`. Furthermore, we identify more examples where the row store achieves competitive result: `Q11`, `Q14`, `Q17`, and `Q19`. However, as we assume, for most queries the row store cannot compete with the column store (cf. Table 6.3; e.g., `Q10`, `Q15`). For `Q6`, we argue that only a minor number of columns has to be accessed and processed (i.e., it is a classical OLAP query), thus, row stores access a lot of unnecessary data. We observe that the Row Store frequently reaches the limits of I/O bandwidth (approximately 120 MB/s) during data acquisition (i.e., read I/O depicted as *red* dashed line; cf. Figure 6.5). We argue that the selectivity of predicates is high enough, thus,

[9]QuadCore @3.33 GHz, 8 GB RAM running an Ubuntu Server 10.04LTS-64bit (2.6.32-41).

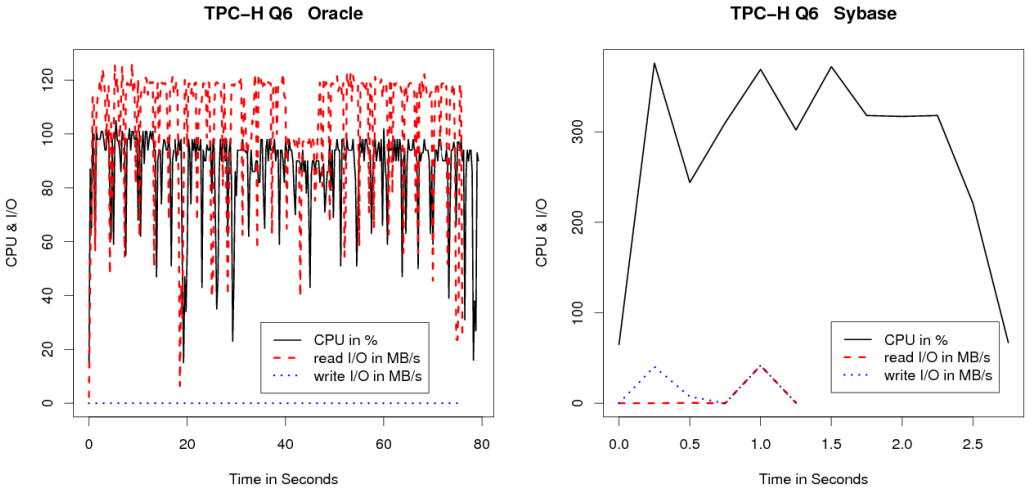


Figure 6.5.: *CPU and I/O for TPC-H Q6 on Oracle and Sybase.*

for both systems, the intermediate results fit into main memory. Hence, we do not observe disk-swapping phases (i.e., write I/O depicted as *blue* dotted line). Sybase has an average CPU utilization (depicted as *black* line) of approximately 274%[10] and 14.19 MB/s (in 1 second 0 to 41.69 MB/s) average disk read for `Q6`; whereas CPUs start processing (67%) after first measurement interval (0.25 seconds) already (cf. Figure 6.5). Oracle's average CPU utilization is approximately 85%, but it needs more than 1 second to start data processing (i.e., waits for data). We observe this behavior via subsequent start of the *black* line for CPU in comparison to the *red* dashed line for read I/O. Furthermore, Oracle's average disk read is 95 MB/s (for 01:19 minutes; cf. *red* dashed line) approximately. These values show how different the amount of data for both systems is to process. In consequence, it is reasonable and inherently that row stores cannot achieve competitive results for this query type.

We can confirm these results for `Q10` (Sybase: ≈ 22 MB/s and ≈ 220%, Oracle: ≈ 92 MB/s (read), ≈ 9 MB/s (write during dump), and ≈ 89% respectively ≈ 12% during dump) as well as for `Q15` (Sybase: ≈ 23 MB/s and ≈ 125%, Oracle: ≈ 98 MB/s and ≈ 87%). Even worse, query `Q5` and query `Q10` (cf. Listings A.18 and A.22 on Page 159 and 160) have long write phases in Oracle (i.e., buffering; cf. *blue* dotted line in Figures 6.6 and 6.7). We argue that the disk swapping decreases overall performance significantly. In contrast, Sybase has shorter write phases (e.g., reconstruct intermediates for final result). These observations correspond to our heuristics (cf. Section 6.2); Oracle uses the whole I/O bandwidth, but cannot utilize all available CPU performance due to the large amount of data and the non-parallel computation. Moreover, main memory is not sufficient for computations in Oracle; even

[10]We state, CPU utilization over 100% indicates that more than 1 CPU core is used.

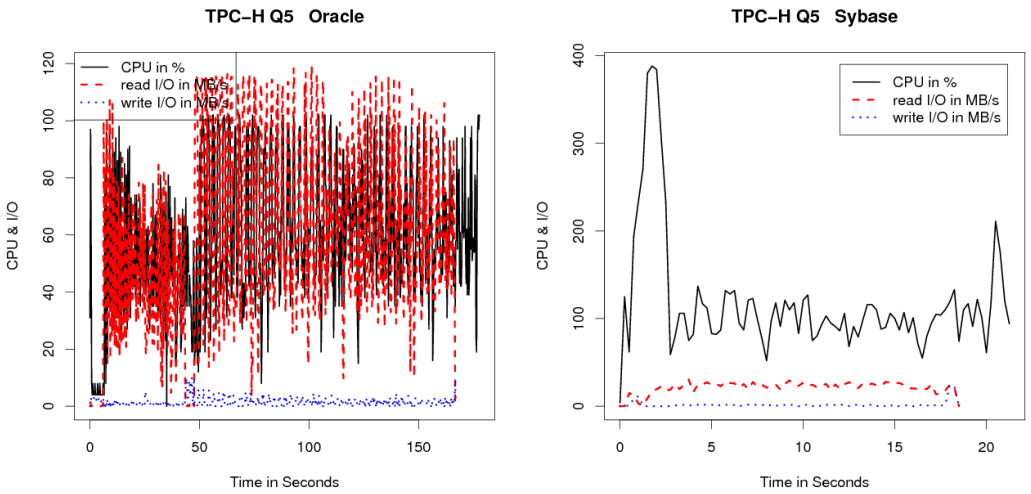


Figure 6.6.: *CPU and I/O for TPC-H Q5 on Oracle and Sybase.*

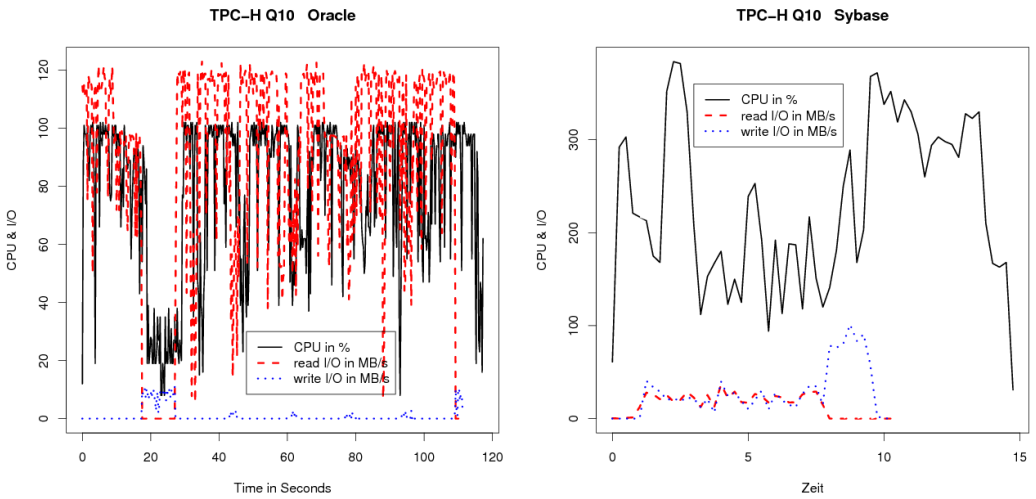


Figure 6.7.: *CPU and I/O for TPC-H Q10 on Oracle and Sybase.*

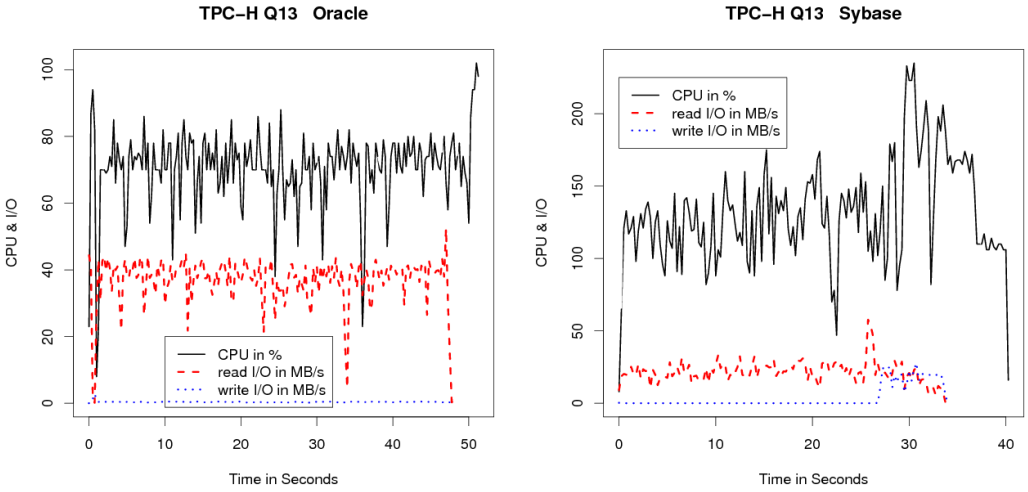


Figure 6.8.: *CPU and I/O for TPC-H Q13 on Oracle and Sybase.*

though main memory is approximately 1/10 of data size. We argue, the selectivity of predicates is as low for `Q5` and `Q10` that the intermediate results do not fit into main memory. These queries are representatives for worse OLAP performance of row stores.

We observe other results for `Q13`, which contains two subsequent groupings on an outer join. For `Q13`, Sybase cannot utilize parallel computation on different columns due to subsequent grouping operations (cf. Figure 6.8) that cause tuple reconstructions. We argue, such behavior is caused by iterative execution of `Q13`. That is, an inner and an outer grouping are computed iteratively (cf. Listing 6.4). For column stores, each grouping causes tuple reconstructions; whereas row stores process as usual and iterate just once more on the intermediate result. We state that such behavior may occur more frequently for complex OLAP queries. We also see this behavior by the CPU usage (approximately 100%). That is, Sybase uses – in contrast to other queries – one CPU mainly. In comparison, the disk usage does not change significantly (approximately 14 MB/s). For Oracle, CPU usage stagnates at approximately 80% and neither I/O bandwidth is fully utilized (approximately 34 MB/s) nor Oracle does swap to disk. For query `Q11`, we observe akin behavior (cf. Figure 6.9) caused by a complex HAVING clause with a sub-query (Sybase: $\approx$ 19 MB/s and $\approx$ 94%, Oracle: $\approx$ 70 MB/s and $\approx$ 94%). The HAVING clause comprises of an aggregation over two columns and a sub-query which selects predicate (cf. Listing A.23 on Page 160). We point in the same direction for `Q14` (cf. Figure A.46 on Page 170), which causes such behavior by subsequent (dependent) aggregations (cf. Listing A.25 on Page 160).

However, we argue that query `Q19` is completely different, because the aggregation is processed over three sets of predicates in disjunctive normal form (cf. Listing A.29 on Page 161). For `Q19`, a huge number of predicate selections, that are not inde-

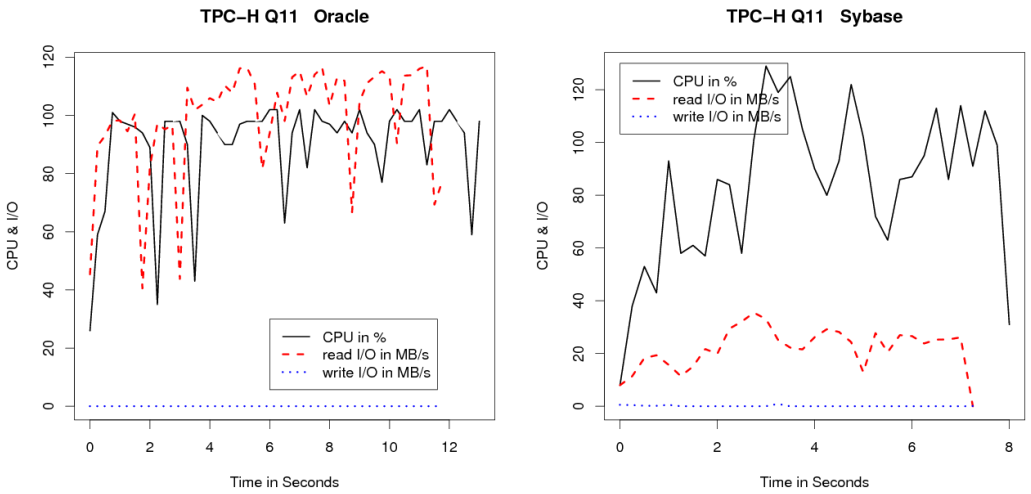


Figure 6.9.: *CPU and I/O for TPC-H Q11 on Oracle and Sybase.*

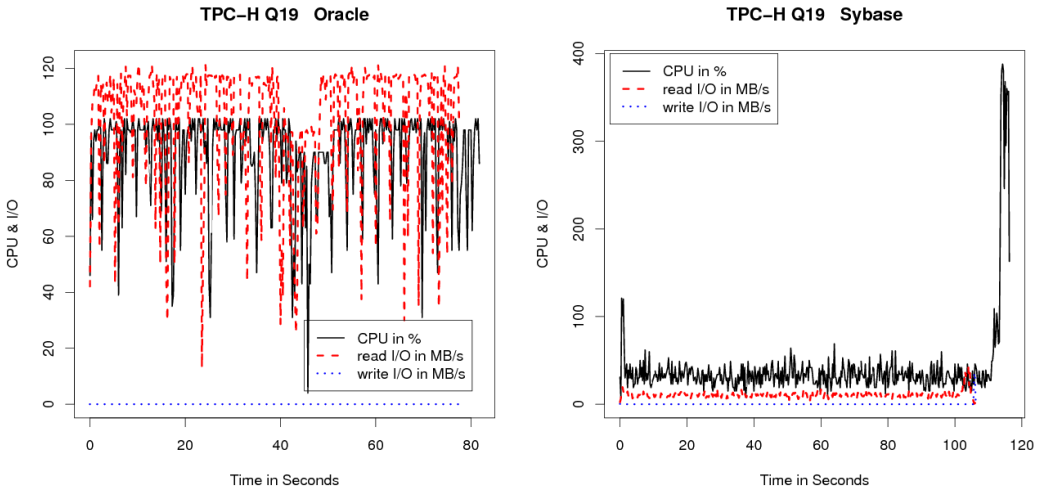


Figure 6.10.: *CPU and I/O for TPC-H Q19 on Oracle and Sybase.*

	Oracle			Sybase		
Tx	**Time**	**CPU**	**I/O**	**Time**	**CPU**	**I/O**
2.4 (NewOrder)	<1	73	2.99	5	65	18.15
2.5 (Payment)	<1	62	29.35	4	75	12.07
2.7 (OrderState)	**<1**	68	4.65	**40**	139	9.05
2.8 (Delivery)	<1	63	8.20	<1	198	11.51

Table 6.4.: *Execution times (in seconds), Ø CPU in %, and Ø I/O (in MB/s) for TPC-C transactions.*

pendent from each other, causes a worse processing scheme (i.e., worse performance) for column stores. Accordingly, column stores have to reconstruct tuples for intermediate results on `LINEITEM` relation (i.e., fact table) as well as on `PART` relation. We observe the interdependence by a scattered computation on low CPU load (cf. *black* and *red* line on a low level in Figure 6.10). We summarize, 17 queries perform better on the column store respectively one on the row stores for TPC-H; whereby four queries show competitive performance on both stores (cf. Table 6.3).

Second, we investigate column-store performance on OLTP workloads. We show that column stores cannot sufficiently support mixed OLTP/OLAP workloads. Consequently, we argue that we need row-store functionality for HDBMSs and HDBSs in this domain. We use the TPC-C benchmark [Fer06] to simulate an OLTP workload. However, we exclude TPC-C transaction (Tx) 2.6 (StockLevel) from our measurements because it is just a cursor declaration for in-memory tasks, and thus, we would see no effect (cf. Listing A.36 on Page 165). We highlight, TPC-C transactions are inherently concurrent. We present the results for our OLTP workload in Table 6.4 for both system (i.e., Oracle and Sybase). For Transactions 2.4 and 2.5 (cf. Listings A.34 and A.35 on Page 163), we observe relatively good performance for column store (i.e., 5 respectively 4 seconds; cf. Figures A.54 and A.55 on Page 172) even though execution time is a multiple of Oracle's execution time (i.e., < 1 second). We argue, the good performance results from the ratio between lookup and write operations[11] (approximately 1 : 1) due to the fact that in general column stores show very good performance on lookups. We can validate our observation – column stores perform well on lookups – with transaction 2.8 that contains lookups only (i.e., two select queries; cf. Listing A.33 on Page 162). Both systems show nearly equivalent performance (i.e., < 1 second; cf. Figure A.56).

In contrast, column stores show poor performance for transaction 2.7 (cf. Listing A.37 on Page 166), that contains twice as many write operations as lookups. Therefore, we observe a more than forty times higher execution time for the column store (i.e., less than 1 second for Oracle and 40 seconds for Sybase). This result cannot be reduced to resource-consumption issues due to the fact that the column store utilizes 139% of CPU and 9.05 MB/s of I/O bandwidth only. We argue that concur-

[11] We state, write operations include inserts and updates.

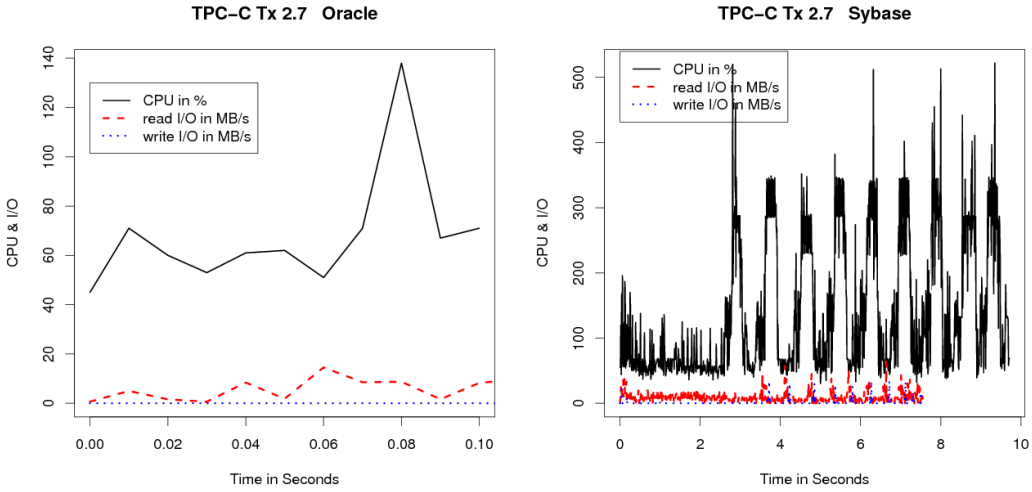


Figure 6.11.: *CPU and I/O for TPC-C Transaction 2.7 on Oracle and Sybase.*

rency control significantly slows down the system (e.g., write operations cause tuple reconstruction for next lookup). The peaks occur while lookups and some inserts on dimension tables are processed in parallel. We see this behavior by the scattered processing scheme (cf. Figure 6.11). We refer to transaction 2.4 where the read-write ratio is approximately 1 : 1 (cf. Listing A.34 on Page163) instead of approximately 1 : 2; whereby the column store just can keep up with the write operations. Our results show that column stores are not suitable for OLTP workloads even when they have read-write storage like Sybase [How10] or C-Store [SAB+05, Aba08]. We argue, write-operation support of column stores is more a functionality completion than focus on OLTP-workload support. We summarize, our results on the TPC-H and TPC-C benchmark underline our heuristics for CPU and I/O consumption (cf. Section 6.2). We argue, the Column Store shows high I/O on TPC-C compared to the Row Store due to reread data for transaction processing. Nevertheless, column stores generally consume more CPU time, whereas row store consume more I/O bandwidth. We refer to Appendix A.3 for queries[12] and transactions that we do not discuss in detail here.

Third, we discuss the impact of mixed workloads to system performance and our heuristics (i.e., behavior on schema that differs from STAR schema). We show our results for TPC-CH queries[13] in Table 6.5. For TPC-CH query `Q5`, we obtain two results for the column store due to the fact that Sybase has issues with modulo computation. That is, we abort the original query after more than 10 hours and restart it without modulo computation. However, our results show that the column

[12] Queries for the TPC-CH benchmark can be found online: `https://db.in.tum.de/research/projects/CHbenCHmark/`.

[13] Queries and source code are available at: `https://db.in.tum.de/research/projects/CHbenCHmark/`.

store cannot significantly outperform the row store for each OLAP query. We argue, the results for both architectures on the TPC-CH benchmark are closer together than for the TPC-H benchmark even though the concurrent OLTP part is not executed during query processing. That is, we observe 13 queries that perform better on the column store and three on the row stores for TPC-CH; whereby five queries show competitive performance on both stores (cf. Table 6.5). We argue, schemas as for TPC-CH are more likely applicable for mixed workload applications than DWH/OLAP schemas (e.g., STAR schema). That is, we may use our observations to improve the overall performance in replicated solutions (cf. Section 6.3).

Query	Oracle	Sybase	Query	Oracle	Sybase
Q1	00:24	00:55	Q12	02:08	00:22
Q2	00:48	00:39	Q13	00:02	00:02
Q3	00:28	00:38	Q14	00:28	00:09
Q4	<00:01	<00:01	Q15	02:06	00:37
Q5	06:10	00:12 (>10h)	Q16	01:33	00:20
Q6	00:23	00:03	Q17	00:46	00:06
Q7	00:25	00:02	Q18	01:28	00:45
Q8	00:01	00:06	Q19	00:04	00:02
Q9	00:41	00:16	Q20	00:58	01:03
Q10	01:10	00:41	Q21	02:14	01:27
Q11	01:05	00:17	Q22	00:12	00:02

Table 6.5.: *Execution times of TPC-CH queries (in mm:ss).*

We conclude that certain OLAP queries are worth to dispatch them to row or column store according to environment and query. Furthermore, OLTP transactions may not dispatched to single column-store environments. We state that mixed OLTP/OLAP workload processing is not competitive on single architecture because row stores do not achieve competitive overall performance on OLAP queries as well as OLAP query performance significantly decreases on column stores due to consistency issues by write operations. Furthermore, we regard these results as conformation for our heuristics. Nevertheless, we discuss the behavior for mixed workloads (including transactions) for (real) hybrid stores in the following.

6.4.3. A Hybrid DBMS Prototype – An Implementation of AQUA²

We discuss results on OLAP, OLTP, and OLTP/OLAP for replicated solutions (i.e., HDBSs) hitherto. In the following, we introduce and evaluate our hybrid solution (i.e., an HDBMS) to dispatch mixed OLTP/OLAP workloads. For a proof of concept and evaluation purposes, we implement an HDBMS based on HSQLDB[14]. HSQLDB is an open-source-row store that is implemented in Java and is widely used (e.g.,

[14] `http://www.hsqldb.org/`.

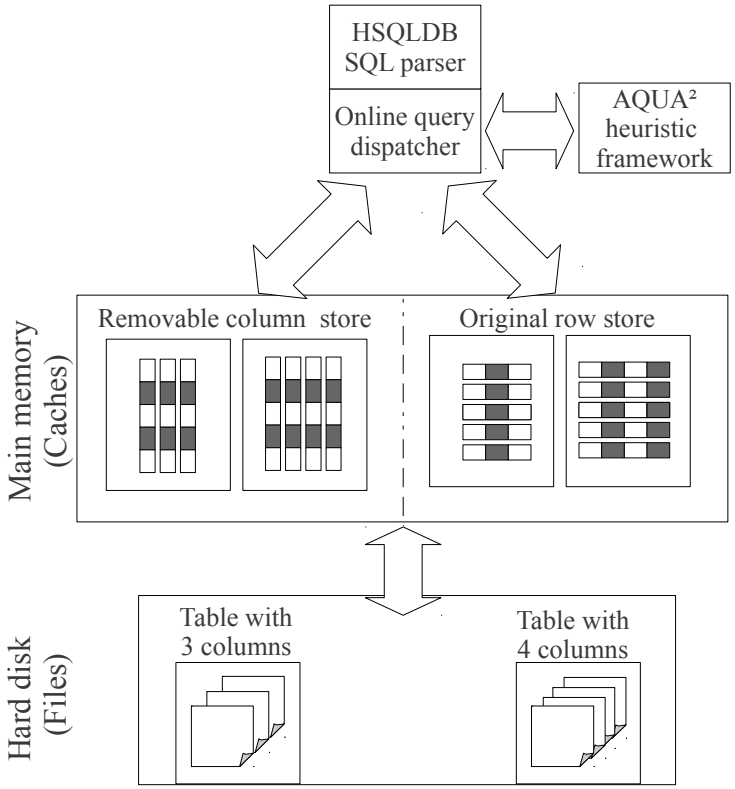


Figure 6.12.: *Hybrid HSQLDB prototype with AQUA²* *integration.*

in Open Office[15]). We implement column-store functionality for HSQLDB based on version 2.2.5 which we briefly present subsequently.

We integrate the column store minimally-invasive to guarantee the comparability of the row and the column store in HSQLDB. That is, we reuse existing functionality as far as possible (i.e., methods and classes). Furthermore, we apply software-engineering methods (i.e., like preprocessors) that allow us to remove our modifications automatically from compiled HSQLDB variants [ABKS13] (cf. Listing 6.13). That is, we are easily able to build and evaluate variants of our prototype (e.g., original row-store implementation). However, we modify and implement *three* core aspects in HSQLDB that are the *persistence* layer, the *caching* behavior, and the *dispatcher* functionality. Our modifications refer to the HSQLDB-table type `TextTable`. We show an overview of modifications in Figure 6.12.

First, we modify the *persistence* layer of HSQLDB. The first modification changes the way how HSQLDB stores tables. That is, we implement a column-wise storage of tables (instead of row-wise). Furthermore, we optimize read-access on these tables to benefit from the column-wise storage on HDD. Originally, the HSQLDB-CSV-table

[15] `http://hsqldb.org/web/openoffice.html`.

```
1  public class QuerySpecification extends QueryExpression {
2          static{
3              String storageType;
4              if(Features.HYBRID_STORE == Features.STORAGE_TYPE){
5                  storageType = "HYBRID_STORE";
6              }
7              if(Features.COLUMN_STORE == Features.STORAGE_TYPE){
8                  storageType = "COLUMN_STORE";
9              }
10             if(Features.ROW_STORE == Features.STORAGE_TYPE){
11                 storageType = "ROW_STORE";
12             }
13             System.out.println("Starting server as inMemory: "+storageType);
14         } ...
```

Figure 6.13.: *Example for features in prototypical implementation of AQUA²*.

manager stores tables row-wise. That is, the table manager administers one file per table in which tuples are organized row-wise. For column-wise access, we change table organization, thus, the table manager administers n files for a table, which consists of n columns. Therefore, we modify the `connect` method in class `TextTable` (cf. Listing 6.14), which handles text-data sources in HSQLDB (e.g., CSV files in our case); while the parent class `Table` remains unchanged and keeps data-structure management and maintenance of tables. With our adaptation, the table manager connects to one file per table column. That is, we add a loop to the initialization of `FilePositions` of a table with i iterations (line 4). The number of iterations corresponds to the number of columns n. Furthermore, we adapt the `TextCache` call to our columnar representation (line 5; cf. Listing A.57). However, our last change in the `connect` method is the incrementation of `nextpos` (line 11) due to the fact that a calculation of positions within a row – as in the original implementation – is not necessary anymore.

In addition, we need to modify the way how HSQLDB reads data from HDD, to prevent unnecessary tuple reconstructions since tuples are read sequentially. Therefore, we modify the internal representation of rows for the persistent storage. The internal data representation is implemented as AVL tree in HSQLDB. Due to the fact that we implement early materialization, we modify class `RowStoreAVLDiskData` only, which implements the persistence layer between internal AVL-tree representation and `TextTable` (text files in our case). The (parent) classes[16] `RowStoreAVLDisk` and `RowStoreAVL` implement the internal representation for cached tables[17] and the table-type independent base implementation of the persistence layer, respectively. However, we overload method `get` in class `RowStoreAVLDiskData` to handle column-wise organized data (cf. Listing A.58). That is, `get` handles row-wise and column-wise organized data dependent on method call. With our `get` method, we read

[16]Note, `RowStoreAVLDiskData` extends `RowStoreAVLDisk`, which extends `RowStoreAVL`. Thereby, `RowStoreAVL` implements the interface `PersistentStore`

[17]Tables that are completely loaded into cache at database start up.

```
1  private void connect(Session session, boolean withReadOnlyData) { ...
2              Row row = null;
3              int nextpos = 1;
4              for (int i = 0; i < this.columnCount; i++) { // number of source files
5                  ((TextCache) cache).cols[i].FilePositions.put(nextpos, 0); // initialize FilePositions
6              }
7              while (true) {
8                  row = (Row) store.get(nextpos, false); // row−wise
9                  if (row == null) {break;}
10                 Object[] data = row.getData();
11                 nextpos++;
12                 systemUpdateIdentityValue(data);
13                 enforceRowConstraints(session, data);
14                 store.indexRow(session, row);
15             }
16     } ...
17     }
```

Figure 6.14.: *Extract of modifications to the data acquisition for the cache (class TextTache).*

data column-wise in chunks (line 7-12) and materialize rows on the level of chunks (line 17-25; i.e., we build currently 100 rows at once). Hence, our prototype reads chunks of each column and then reconstructs tuples if necessary which results in better cache utilization and a reduced CPU overhead. For the implementation, we use the `RowInputInterface`, which we discuss with modifications for caching in the following.

We modify the *caching* behavior, which is our second core aspect of implementation, because there is only row-wise caching present in HSQLDB. Additionally to the row store cache, we implement a cache for the column store in HSQLDB. Therefore, we modify the way how data files are opened and read into cache (class `TextCache`; cf. Listing 6.15). At the beginning, we adapt the buffer initialization (`initBuffers`), thus, n columns of a table (i.e., files) lead to n initialized buffers (cf. line 8-15). Furthermore, we change method `readObject` to `myReadObject`; whereby both methods read a (text) stream of characters. Both methods search for special characters (e.g., quotation marks, line endings) for processing purposes (cf. line 30-46). Thereby, the end of a row is identified by line endings (i.e., \n or $c = 10$; line 35). In contrast to the original method, `myReadObject` processes on chunks of rows (i.e., column values). This way, we prevent complete cache-hierarchy calls for each column value. Therefore, we introduce `CHUNKSIZE` and `chunkPosCount` to iterate over data (line 6 and 21). With these variables, we control processing of a complete chunk (here 100 values or rows) without method recall (line 27-48). We additionally reduce overhead by skipping of empty rows (line 38). If a chunk is completely processed, we pass data to the `RowInputTextInterface`[18] and return the result (line 52-57).

In addition, we add a method `getFromFile` (cf. Listing A.59 on Page 176), which implements the transfer from persistent storage to cache. This method takes over

[18]Note, `RowInputTextQuoted` extends `RowInputText`, which implements `RowInputTextInterface`.

```
1  import columnar.MyColumn; ...
2  import org.hsqldb.RowAVLDiskData; ...
3  public class TextCache extends DataFileCache { ...
4      public MyColumn[]  cols;
5      public RowInputInterface[]  rowIns;
6      public static final int  CHUNKSIZE = 100; // constant size for now
7      ...
8      protected void initBuffers() {
9          int anzahl = this.table.getColumnCount();
10         if (isQuoted || isAllQuoted) {
11             rowIn = new RowInputTextQuoted(fs, vs, lvs, isAllQuoted);
12             rowOut = new RowOutputTextQuoted(fs, vs, lvs, isAllQuoted,stringEncoding);
13             for (int i = 0; i < anzahl; i++) {rowIns[i] = new RowInputTextQuoted(fs, vs, lvs, isAllQuoted);}
                 // number of columns
14           } ... // else looks alike
15     } ...
16     protected RowInputInterface[] myReadObject(int rowPos) { // iteration for chunks
17         for (int i = 0; i < cols.length; i++){rowIns[i] = myReadObject(rowPos, i);}
18         return rowIns;
19     }
20     protected RowInputInterface myReadObject(int rowPos, int colIndex) { ...
21             int chunkPosCount = 0; // counter for current position in chunk
22             if (cols[colIndex].FilePositions.containsKey(rowPos) == false) {return null;}
23             int pos = cols[colIndex].FilePositions.get(rowPos); int oldPos = pos;
24             pos = findNextUsedLinePos(pos, colIndex);
25             ...
26             dataFiles[colIndex].seek(pos);
27             while (!complete) { ...
28                 c = dataFiles[colIndex].read(); // added [colIndex] // single character in int representation
29                 ...
30                 switch (c) { ...
31                     case CR_CHAR : //13 "\r"
32                         c = 10; // character represented as int --> "\n"
33                         if (chunkPosCount >= CHUNKSIZE - 1) {c = 13;wasCR = !hasQuote;} // "- 1"
                             because not incremented yet
34                         break;
35                     case LF_CHAR : //10 "\n" // end of row --> abort; complete = true!
36                         chunkPosCount++; c = 124; //124 // separator "|" := 124
37                         pos = (int) dataFiles[colIndex].getFilePointer(); // position to current FilePointer
38                         pos = findNextUsedLinePos(pos, colIndex); // skips empty rows --> deleted rows
39                         if (pos == -1) {c = 10;complete = !hasQuote;} // <100 values --> abort!
40                         else {cols[colIndex].FilePositions.put(rowPos + chunkPosCount, pos);}
41                         dataFiles[colIndex].seek(pos);
42                         if (chunkPosCount >= CHUNKSIZE) {c = 10;complete = !hasQuote;} // read 100
                             values resp. CHUNKSIZE
43                         break;
44                     default : ... // proceed --> next
45                 }
46                 if (c != 10 || chunkPosCount >= CHUNKSIZE) {buffer.append(c)}; // != "\n"
47             }
48             if (complete) {
49                 int length = (int) dataFiles[colIndex].getFilePointer() - oldPos;
50                 if (wasNormal) {length--;}
51                 if (rowIns[colIndex] == null) { // fs, vs, lvs, isAllQuoted --> DB initialization parameters
52                     if (isQuoted || isAllQuoted) {rowIns[colIndex] = new RowInputTextQuoted(fs, vs, lvs,
                         isAllQuoted);}
53                     else {rowIns[colIndex] = new RowInputText(fs, vs, lvs, isAllQuoted);}
54                 }
55                 ((RowInputText) rowIns[colIndex]).setSource(buffer.toString(), pos, length, rowPos); // 100
                     values pipelined; added [colIndex] and rowPos
56                 return rowIns[colIndex]; // materialization
57             } ...
```

Figure 6.15.: *Extract of modifications to the data acquisition for the cache (class TextCache).*

responsibility for locking and data acquisition of data, which is not resident in the cache, from the corresponding original `get` method; whereas other `get` methods remain unchanged. In contrast to the original `get` method, `getFromFile` process column-wise and reads chunks of data (using `RowStoreAVLDiskData`). However, we implement a fully redundant in-memory caching for each table (i.e., row and column representation). We use the redundant caching approach to dispatch workload queries. We state that HSQLDB keeps tables in cache as long as there is sufficient main memory. Note, for evaluation purposes, we need to exclude effects caused by HDD access (especially when only one of the representations is swapped to disk) and therefore, we forbid swapping of (parts of) the caches to HDD. As a result, query processing of our prototype works fully in main memory for now. On the one hand, main memory processing (after data acquisition) eliminates unintended side effects and improves the accuracy of measurement. On the other hand, we abstract from many real-world scenarios with this assumption. Nevertheless, we need the first modifications – addressed above – for initial loading and persistent storage of tables.

Third, we implement the *online query dispatcher* based on our heuristic decision framework for HDBSs respectively HDBMSs (cf. Section 6.2). We integrate the online query dispatcher into an HDBMS to demonstrate benefits of hybrid solutions. Currently, we utilize a subset of our heuristic-decision rules. As general rule of thumb, OLAP queries are processed on the column store and OLTP transactions on the row store. We implement an alternative data-flow path to process queries on the column store. In consequence, we access both stores for data processing and load data into both in-memory representations. For the alternative data-flow path, we modify query processing slightly. We reuse functionality of the SQL parser, which extracts expression from SQL text and maps them to different expression groups (e.g., subquery). This functionality makes possible query categorization via Boolean variables (e.g., `isAggregated`; cf. line 5 in Listing 6.16). That is, we dispatch (execution of) a query based on contained expressions as method `buildResult` is requested. In our example (cf. Listing 6.16) for the hybrid store, we check for queries without aggregates (line 5) that are executed on the original data-flow path (i.e., row-wise; line 6-8); whereas queries with aggregates are executed on the alternative data-flow path (i.e., column-wise; line 10-23). For comparison, we present a code snippet for the Row Store and Column Store configuration in Listing A.60 (on Page 176). We present additional methods in Appendix A.5 (e.g., for predicate evaluation), which are adapted for columnar behavior.

We argue, first results on loading behavior of our column-store implementation go into the same direction as our results on TPC-C transactions in Section 6.4.2. Loading the TPC-H benchmark (i.e., inserts; scale factor 0.01) takes twice as long as the original row-store implementation. Furthermore, we observe higher main-memory consumption for our column-store implementation which is contrary to other column-store implementations. We argue that the higher consumption is caused by: (a) storage overhead due to additional partitioning information and (b) lack of data compression that catches up the partitioning overhead in other column-store implementations. Finally, we implement early materialization in our first prototype. We

```
1  private Result buildResult(Session session, int limitcount) { // called once per query
2              ...
3              RangeIterator it = rangeIterators[currentIndex];
4              if(Features.STORAGE_TYPE == Features.HYBRID_STORE) {
5                  if (!isAggregated || cache[0]==null ) {
6                      if (it.next()) { // it.next fetches next row which satisfies conditions
7                          if (currentIndex < rangeVariables.length − 1) {currentIndex++; continue;}
8                      } else {it.reset(); currentIndex−−; continue;}
9                  }
10                 else { // isAggregated
11                     if (it.myNext(cache, rowPos, count, currentIndex)) { // condition check and it.myNext
                           fetches next column value (row of column)
12                         if (currentIndex < rangeVariables.length − 1) {currentIndex++; continue;}
13                         if (rowPos[currentIndex] > count[currentIndex]) {
14                             if (rowPos[0] >= count[0]) break; it.reset(); rowPos[currentIndex] = 1;
                               currentIndex−−;
15                             if (currentIndex >= 0) rowPos[currentIndex]++; continue;
16                         }
17                     } else {
18                         if (rowPos[0] >= count[0] +1) break; it.reset(); rowPos[currentIndex] = 1; // set
                           outer index to 0
19                         currentIndex−−;
20                         if (currentIndex >= 0) rowPos[currentIndex]++; continue; // increase inner index
21                     }
22                 }
23             } ... }
```

Figure 6.16.: *Query dispatching for aggregates in hybrid store configuration (class QuerySpecification).*

are aware of the fact that early materialization is not as competitive as late materialization in column stores [AMH08]. Consequently, we are aware that our prototype has performance issues, thus, we do not expect outstanding but competitive results.

6.4.4. Evaluation – Hybrid Solution

We introduced previously our HDBMS prototype that we use to validate our claim: hybrid stores can achieve better performance for mixed workloads. For our evaluation, we use the TPC-CH benchmark [CFG+11] which is basically a mixture of the analytical TPC-H and the transactional TPC-C benchmarks. Therefore, we use a small version of TPC-CH with 100MB of data (i.e., 1 warehouse) due to the fact that HSQLDB is not designed for big-data applications. Nevertheless, we argue that the amount of data is sufficient for a proof of concept. We use a row store (i.e., original HSQLDB) and our prototype (i.e., an HDBMS) for the evaluation.

We perform all experiments on the same computer (Intel Core i7, 8 GB RAM, Windows 7-64bit, Oracle Java Virtual Machine JRE 1.6, maximum heap size 1.5 GB). For each variant, we use two workload queues querying the database in parallel. The first workload queue comprises of OLAP queries; the second one comprises transactional load of the benchmark using the `read-committed` isolation level [SHS05, Pages 600 ff.]. We perform the experiment 120 times. Furthermore, we compute a robust mean value using a γ-trimming approach ($\gamma = 10\%$) to compensate outliers and ensure

Query	Row	Hybrid
Q1	7,183.85	6,780.90
Q4	15.11	14.62
Q6	3,134.69	5,763.29
Q12	1,027.07	29.21
Q16	33,223.67	15,591.78
Q19	24,842.66	1,967.64

Table 6.6.: *Ø Execution times (in ms) for TPC-CH queries.*

statistical soundness. We limit the number of queries to those that probably cause interesting insights. As a result, our test program uses TPC-CH queries `Q1`, `Q4`, `Q6`, `Q12`, `Q16`, and `Q19`[19]. We select these queries due to the fact that TPC-H queries `Q1`, `Q12`, and `Q16` are close together; whereas `Q4` and `Q6` give advantage to column stores respectively `Q19` gives advantage to row stores (cf. Table 6.3). For TPC-CH, queries `Q4` and `Q19` show competitive results on both stores; whereas `Q6`, `Q12`, and `Q16` give advantage to column stores respectively `Q19` gives advantage to row stores (cf. Table 6.5). Finally, we remove data changes that are caused by the benchmark, after each pass of our experiment from the table space. That is, we reset the table space to ensure comparability of single passes.

We summarize our results (i.e., the average execution time) for the respective TPC-CH queries with regards to our hybrid prototype and the row-store variant of HSQLDB in Table 6.6. Our results show that the hybrid solution is faster for the OLAP queries – using the column-store part – except for query `Q6`. The result for `Q6` is reasonable due to the fact that we implement early materialization for the hybrid solution. That is, the hybrid has to reconstruct a large amount of tuples, that are composed of three columns from a relatively large table (i.e., `ORDERLINE` – second largest in benchmark), to compute predicate selection on two columns and summation on the third column. Nevertheless, the hybrid solution significantly achieves the better overall performance for mixed workloads. We argue, our results are sound and significant (e.g., hybrid solution is more than 12 times faster for `Q19` respectively more than 30 times for `Q12`).

We make additional analysis to assign execution times to single high-level operations (i.e., database operations) to gain further insights and explain our observations. For our analysis, we use the Java profiler `VisualVM`[20] which is part of Oracle's Java Development Kit. The results of this analysis for query `Q6` – where row-store is faster – and query `Q19` – largest time benefit of hybrid – are depicted in Table 6.7.

For query `Q6`, we observe savings in CPU consumption for predicate selection and aggregation on the hybrid solution. That is, CPU consumption drops from 72.2% to 54.9% for predicate selection respectively from 21.1% to 4.7% for aggregation (i.e.,

[19]Note: TPC-CH queries are akin to TPC-H queries but not equivalent.
[20]`https://visualvm.java.net/`.

sum). Nevertheless, we observe a very expensive operation – the tuple reconstruction – for query processing in the hybrid store that utilizes the column-store part, to evaluate the selection predicates for aggregation. That is, the column-store part in the hybrid solution causes 40.1% CPU consumption for this operation (cf. Table 6.7). As query processing in HSQLDB is optimized for row stores, the selection predicates are evaluated tuple-wise. Consequently, the query processor reconstructs tuples and then accesses necessary attributes; whereby evaluation of predicates is performed tuple-wise. We state that this behavior is inherently due to our minimally-invasive integration with early (tuple) materialization. However, whenever a tuple is reconstructed with necessary attributes and these attributes are accessed, cache hierarchies can be used efficiently since all values are stored in a tuple and thus, close to each other. In contrast, when we apply the column-store part of the hybrid approach, Java cannot utilize caches as efficient to reconstruct tuples. `VisualVM` confirms our expectation according to computational overhead by tuple reconstruction as almost half CPU time (40.1%) is used for tuple reconstruction during computation of `Q6`. Nevertheless, the difference in execution times between both variants is not that high (i.e., 3,134.69 ms for row store and 5,763.29 ms for hybrid solution) because aggregation itself is very efficient since Java uses the cache here efficiently (21.1% CPU consumption for row store and only 4.7% for hybrid solution).

	Query Q6		**Query Q19**	
Operations	**Row**	**Hybrid**	**Row**	**Hybrid**
Selection	72.2	54.9	13.1	16.3
Aggregation	21.1	4.7	–	–
Join	–	–	72.2	77.8
Tuple reconstruction	–	40.1	–	<5

Table 6.7.: *Operations with more than* $\approx 5\%$ *avg. CPU Consumption (in %).*

For query `Q19`, we observe the highest time benefit of the hybrid solution compared to the row store. For this query, tuple reconstruction is not crucial for two reasons even though two large tables are involved (i.e., second and fourth largest in benchmark). First, HSQLDB utilizes indexes and does not scan whole relations to evaluate join partners of both tables (i.e., `ORDERLINE` and `ITEM`). Hence, the column-store part benefits from the implementation of indexes in HSQLDB. In the original row-store variant indexes contain precomputed row iterators for fast iteration over whole indexed data that needs to be updated because of TPC-CH-write load. In contrast, the column store only caches these iterators that contain information how rows be reconstructed (i.e., like bitmaps), and thus, there is less computational overhead which leads to faster execution times. Second, the query result is an aggregate (i.e., sum). Consequently, the hybrid solution reconstructs tuples with necessary attribute for `ORDERLINE` and `ITEM` and computes the predicate selection like the row store afterwards. Furthermore, the hybrid solution computes the join with the above

mentioned benefits; whereby the aggregation is computed on-the-fly due to the fact that the respective attribute is an independent part[21] of the intermediate result (i.e., join result) and no further tuple reconstruction is needed. We argue, such behavior causes the absence of measured values for aggregation for both variants (cf. Table 6.7). As a result, the disadvantage of tuple reconstruction is overlaid by other operations and thus, negligible because the consumed CPU time is less than 5% in contrast to 40.1% for TPC-CH query `Q6`.

We conclude that our hybrid solution – an HDBMS – shows at least competitive and in some extent outstanding results for mixed OLTP/OLAP-workload processing, even though our dispatching approach is very rudimentary. That is, our assumption in Section 6.1.1 proved to be true that hybrid stores are more efficient on mixed workloads than each architecture alone is. We suggest further implementations for HDBMSs based on our results due to the fact that our implementation lacks of (columnar) compression, particular vector-operations support, late materialization and correspondingly columnar processing, etc.

6.4.5. Discussion on Improvements for AQUA²

We presented the results of our experiments; whereby we pointed out some weaknesses and deficiency. In this section, we discuss improvements and further ideas.

We showed the suitability of our approach on redundant data storage (cf. Sections 6.4.2 and 6.4.4. However, we are aware that fully redundant storage cause several problems (e.g., storage-space consumption, consistency). We assume that our approach is also feasible for non-redundant hybrid stores we mention before (e.g., time-bound data partitioning) (cf. Section 6.3). Another opportunity to reduce redundancy in hybrid stores is, we observe which part of the database schema performs with respect to given workloads best, and thus, we allocate schema parts to different stores. In Chapter 5, we introduce a storage advisor for relational DBMS. Whether we reduce redundancy in hybrid stores based on time-lines or based on performance, we may use our storage advisor for performance evaluation for physical (static) schema design following other advisors [ZRL+04, BC07] for row stores. We further argue, the advisor approach can be used to monitor systems and alerts on diminishing performance at threshold exceeding (i.e., alerter approach; e.g., by Bruno and Chaudhuri [BC06]), thus, we periodically redesign store-spanned schemas. In consequence, we come straight to the point that we extend ideas of Agrawal et al. to automatic physical design [ACN06] for systems with several storage types. Therefore, we have to extend our heuristic-based query dispatching by information on data distribution/locality. Moreover, we have to combine both the storage advisor and the online query dispatcher for dynamic behavior (e.g., evolutionary schema). This approach is feasible on different granularities of distributed storage. Nevertheless, fine-grained distribution can cause high reorganization (i.e., on changes) and administration cost. We suggest that an approach on table level may be a good trade-off

[21] We note, predicate selection is done before join, thus, there is no further dependency for the aggregate.

between benefit and administrative overhead. We are careful to point out that these ideas demand for a store-spanning query engine due to the fact that intra-query parallelization and intermediate-result exchange have to be supported. That is, such query engine is dependent on the underlying stores. We argue, a query engine for distributed processing across different contradicts our current vision of a transparent hybrid query interface (cf. Section 6.3).

We further argue that our approach may be improved without a store-spanning query engine. We suggest the implementation of further heuristics which do not demand for such query engine (e.g., for complex OLAP queries or to avoid load peaks on redundant storage; cf. Section 6.2). Additionally, we may improve our solution by implementation of other column-store-typical approaches and usage of a more sophisticated column-store implementation. We state, implementation of late materialization and thus, columnar processing improves performance for several OLAP queries (e.g., for `Q6`). We assume that support of both materialization methods may be promising (cf. Section 3.1); whereby the most suitable is chosen based on query structure. Furthermore, we should implement (columnar) data compression, compressed data processing, and vector-operation support (cf. Section 6.4.4) which are essential for column-store performance and inherently belong to column stores.

6.5. Summary

We presented our approach – a hybrid query interface – for mixed OLTP/OLAP workloads on hybrid stores here in Chapter 6. Therefore, we used our approaches concerning architecture-independent workload statistics and storage-architecture decision based on these statistics.

We integrated our previous work into a hybrid query interface named Automated Query Interface for Relational Architectures (AQUA$^{\mathbf{2}}$). That is, we combined previous work in the *offline-decision framework* and used insights therefrom to build an *online query dispatcher* for mixed workloads on hybrid stores (cf. Section 6.1.1). We refered to two solutions – the replicated solution and the hybrid solution – in our approach; whereby both solutions maintained data fully redundant in two stores (i.e., column and row store). The replicated solution was composed of two DBMS (i.e., an HDBS); whereas the hybrid solution was an HDBMS that combined both stores in one DBMS. In the following, we considered different optimization approaches and levels (cf. Section 6.1.2). We discussed global (i.e., in our approach) and local optimization (i.e., at the store) as well as where the usage of rule- and cost-based optimization would be suitable. We stated that rule-based optimization was most suitable for global optimization due to computational cost of cost-based optimization and transparency ideas concerning our query interface. Furthermore, we argued that usage of rule- and/or cost-based optimization belongs to the store at local level.

We presented insights into design and query processing on HDBS and HDBMS to establish a rule-based optimization approach in Section 6.2. Therefore, we introduced a set of rules that contains rules for inter- as well as intra-query parallelization.

We implemented inter-query-parallelization rules in our query dispatcher. However, we discussed further query-dispatching criteria that were focused on load balancing (e.g., throughput maximization) and time-bound queries (e.g., near real-time; cf. Section 6.3). We had to adapt several approaches from distributed DBMS to implement such features; whereby we focused on a transparent approach for now.

In Section 6.4, we performed a performance evaluation for both solutions (i.e., replicated and hybrid). We performed the TPC-H and TPC-C benchmark on the replicated solution. As a result, we observed that neither system outperformed the other for both benchmarks; whereas the row store dominated as anticipated the TPC-C benchmark and the column store dominated the TPC-H benchmark. Hence, we performed the TPC-CH benchmark and concluded that the performance was closer together than for the other benchmarks. That is, we saw our assumptions confirmed for rule-based query dispatching. Moreover, we implemented a prototype in HSQLDB to validate our considerations in an HDBMS (cf. Section 6.4.4). We choosed a minimally-invasive approach that included a second persistence approach as well as a second cache (i.e., both columnar organized). Furthermore, we implemented our heuristic framework that dispatched queries between both variants (i.e., the original data-flow path and the new columnar data-flow path). We performed the TPC-CH benchmark on our prototype as well as on the original HSQLDB. We observed competitive and in some extent outstanding results for our prototype. That is, our prototype outperformed the original HSQLDB implementation. Subsequently, we discussed the impact of tuple reconstruction based on CPU consumption for our prototype. Finally, we discussed possible improvements for our approach (cf. Section 6.4.5) that implied a more fine-grained heuristic implementation, further implementation of typical column-store features, and redundancy reduction in HDBMSs.

7. Related Work

We consider related work to our approaches in this chapter. Therefore, we compare related approaches to partial aspects of our work as well as to the overall idea for hybrid (relational) data management. We compose our considerations in comparison to five different aspects. First, we consider workload-decomposition and statistics-analysis approaches with respect to our workload-decomposition approach (cf. Chapter 4) and second, we consider advisors for physical design in relational DBMS (cf. Chapter 5). Third, we discuss relevant ideas in the field of self-tuning, especially in terms of physical design (cf. Chapters 5 for cost calculation and 6 for implementation). Fourth, we discuss architectural approaches for relational data management especially Column Stores and OLAP[1]. Finally, we consider (hybrid) approaches, which focus on more generalized DBMSs, and try to overcome architectural drawbacks for mixed requirements (e.g., near real-time OLAP; cf. Chapter 6).

7.1. Related Work on Workload Decomposition & Representation

Query optimization as well as physical design has to decompose workloads for optimal query-plan respectively physical-design estimation. Therefore, researchers propose several approaches with respect to different optimization criteria that we briefly introduce and compare with our approach in the following (cf. Table 7.1).

An approach for optimal disk performance on mixed workloads proposed by Turbyfill [Tur88] characterizes (disk) access pattern for OLTP, batches, and ad-hoc queries[2] based on heuristics (i.e., experiments). The access patterns are used to aggregate similar workload parts and to decide how these patterns are executed – (a) in parallel with possible drawbacks to OLTP or (b) in different time windows.

In contrast to the heuristic-based approach by Turbyfill, Raatikainen [Raa93] discusses general hardness of automatic workload classification and of cluster analysis, which arise from classification. Raatikainen argues that (statistical) workload classification is not necessarily feasible for query-optimization purposes. On the one hand, classification is highly dependent on similarity function for clusters. On the other hand, stability and validity of clusters are identified as an issue. Finally, Raatikainen shows instability of response times for clustering algorithms, thus, such an approach is not suitable for our purpose – at least in terms of query processing due to time constraints and their volatility.

[1] In Chapter 2, we give a coarse overview for corresponding approaches in Row Stores.
[2] Nowadays, batches and ad-hoc queries summarized as OLAP.

Nevertheless, Chaudhuri et al. use a clustering approach which targets on superior support of self-managing DBMSs by reduction of size and diversity of workload information [CGN02]. The used distance function is focused on index selection in this approach (i.e., to reduce complexity for the index selection problem). However, workloads have to be known in advance. Whenever a workload is not known in advance, Chaudhuri et al. use sampling approaches additionally (i.e., additional complexity for quality results). Holze et al. also target on the reduction of size and diversity of workload information. Therefore, they use a clustering approach based on the generalized Minkowski metric to evaluate the similarity [HGR09]. In more detail, Holze et al. generalize the approach of Chaudhuri et al. [CGN02] for ongoing workloads (i.e., without sampling). Both approaches concern the problem of cluster stability stated by Raatikainen. That is, the minimal number of cluster k is determined by a quality loss measure for new clusters that in total must be below threshold δ. Chaudhuri et al. solve the validity problem by the assumption that workloads are known in advance respectively sampling; whereas Holze et al. solve this problem with a learning phase. That is, cluster adaptation freezes after learning phase that ends whenever the maximum distance of new elements to existing clusters fall short of threshold δ. Hence, periodical execution of cluster analysis is limited to an applicable amount by user interaction at expense of cluster quality.

Another point of view for workload decomposition is the temporal aspect. The temporal aspect is suitable whenever different workload types occur periodically (e.g., OLAP and ETL, daily operations and reorganizational tasks). That is, decomposed workload parts represent similar workload tasks for a period of time, thus, optimization (e.g., physical design) is focused on workload parts for a corresponding time period instead of optimization for (*24/7*) overall workload. Agrawal et al. name such temporary workload parts with the term *sequence* (i.e., a set of SQL statements). The following two points are unique characteristics for this approach: (a) it does not distinguish between different physical design approaches (e.g., indexes, materialized views) and (b) it preserves query respectively (SQL)-statement order.

In contrast to the previously introduced approaches, the following two approaches focus on OLAP respectively DWH workloads. That is, the other approaches are not limited to or are at least not especially designed a single for workload domain. Ghosh et al. present a clustering-based decomposition in which template-query plans represent workload clusters [GCCK07]. A template-query plan is a query plan in which all operations remain but identifying values (e.g., table name) are removed (like a query with bind variables). A template-query plan represents all queries within a certain distance (i.e., a cluster); whereby a hierarchical similarity approach – called *SIMCHECK* – is used to reduce complexity. Query plans are reused in this approach whenever a similar cluster exists (i.e., a template-query plan). As a result, Ghosh et al. reduce query-optimization cost for complex queries greatly. Nevertheless, template-query plans are sub-optimal for similar queries but according to the authors fairly close to optimal. An approach by Favre et al. [FBB07] is designed for workload evolution (i.e., the queries to be analyzed) in DWHs. That is, Favre et al. do not focus on workload analysis for physical design, but they focus on query

sets, which represent workloads after evolution. Therefore, they analyze updates on data and schema to update the workload (i.e., add, delete, or adapt queries).

One may be surprised at our set of related approaches. To the best of our knowledge, besides our approach exists no approach that supports workload decomposition and representation for physical-design estimation and query optimization. Therefore, we present related decomposition approaches for query processing (e.g., by Ghosh et al. [GPSH02]) as well as for physical design (e.g., by Chaudhuri et al. [CGN02]).

We consider the access-pattern approach for mixed workloads by Turbyfill [Tur88] due to the fact that our approach inherently considers optimal access (patterns) by storage-architecture selection respectively query dispatching to different storage architectures. In more detail, we also consider disk-access patterns for mixed OLTP/OLAP workloads because our approach is not limited to main-memory DBS. The heuristic-based approach for workload patterns is related to our heuristic framework for physical design and query processing. In contrast to our approach, this approach is limited to a certain DBMS (type). We additionally provide iterative improvement for our framework.

Furthermore, we discuss the considerations of Raatikainen [Raa93] to show hardness of workload clustering (cf. Lange et al. [LRBB04]; e.g., cluster may be found where no (natural) cluster exists). In line with Raatikainen, we argue that clustering is not suitable for workload representation in general. On the one hand, we state that clustering algorithms are too complex for a multi-purpose approach. That is, identification of optimal number of cluster is hard, even if workload is not known in advance (cf. Chaudhuri et al. [CGN02]), but cluster stability and validity are more difficult for ongoing workloads (cf., by Holze et al. [HGR09]) that implies costly cluster recalculation (e.g., by user interaction). Due to the extension of ideas by Chaudhuri et al. [CGN02], the approach by Holze et al. [HGR09] is feasible for OLTP and OLAP in a broad outline. However, we argue that workload clustering is not cost-efficient for query optimization, but we do not prevent workload clustering with our approach. Clustering on top of our workload-representation approach is still applicable even though our superjacent approaches may be adapted. On the other hand, clustering hides the workload sequence, which is crucial for design tasks and query processing in mixed OLTP/OLAP in our opinion.

In line with Agrawal et al. [ACN06], we argue workload sequences are an important aspect for quality of physical design due to the fact that requirements and/or drawbacks for special tasks (e.g., ETL) are hidden in overall workloads. That is, tuning aspects are not considered that can have crucial impact on performance of a workload sequence (e.g., indexes not dropped before ETL/reorganization). The approach is designed for, but not limited to, DWH environments – we argue, workload sequence with variable requirements can arise anywhere (not for DWHs only). In comparison, our approach is not limited to certain domain and not focused on access structures, but preserves order of SQL statements as well (i.e., the sequence).

Another approach also targets on DWHs especially on complexity reduction of workload representation for query processing is proposed by Ghosh et al. [GPSH02]. The approach reduces query-evaluation time (i.e., for query plan creation) for sim-

Approach	OLTP/ OLAP	Arbitrary Cost Model	Preserve Query Order	Workload Aggregation	Architecture Independent
Turbyfill [Tur88]	•	○	○	•	○
Raatikainen [Raa93]	–	○	○	•	–
Evrendilek and Dogac [ED95]	–	○	–	○	•
Chaudhuri et al. [CGN02]	–	○	○	•	–
Agrawal et al. [ACN06][3]	–	○	•	•	•
Ghosh et al. [GCCK07]	○	–	○	•	○
Favre et al. [FBB07]	○	○	○	○	•
Holze et al. [HGR09]	•	○	○	•	•
AQUA^2	•	•	•	•	•

Table 7.1.: *Comparison of key aspects for workload decomposition and representation. Legend: • fulfilled, ○ not fulfilled, – no information available.*

ilar queries. They focus on complex OLAP queries where query-evaluation can be notable. That is, they reduce query-evaluation time by orders of magnitude, but evaluation time is in the range of milliseconds and below even without optimization. We assume, benefits for query evaluation are less significant for less complex queries. We argue, query execution on the most suitable architecture with optimal query plan is more crucial than savings in query-evaluation time. However, we do not directly support reuse of query-plans. As mentioned before for other clustering approaches, clustering for complexity reduction still is applicable on top our representation.

The approach of Favre et al. [FBB07] is located in the DWH domain as well. Their approach solves issues for the correctness of workload information induced via DDL and DML in ETL. In other words, the approach prevents usage of old invalid workload information for workload analysis. In our domain, schema modifications in a bulk are not common, thus, a complex analysis for schema evaluation is not necessary. However, the basic idea is of interest for automatic redesign in hybrid systems. Nevertheless, evolution of workload information (e.g., column renaming) can be easily automated via DDL monitoring or recalculation of DBMS statistics via DML monitoring (i.e., aging) which is only necessary for reusing query-plans.

We summarize key aspects for the above considerations in Table 7.1. We highlight that in contrast to our approach, neither approach supports arbitrary cost models due to specialization of approaches for a certain domain in one architecture even though some approaches are not architecture-dependent directly (e.g., [ACN06, HGR09]).

Finally, we consider an approach, which is marginally related only, by Evrendilek and Dogac [ED95] for two reasons. First, they propose a decomposition approach for distributed processing even though the approach is focused on queries. Second, the approach follows similar ideas for (sub-) query distribution including availability of cost estimates at global level. Both aspects are related to our hybrid-system approach. Evrendilek and Dogac propose a cost model for query distribution in distributed DBMSs with respect to data fragmentation and highlight the idea of

[3]Designed for DWHs, but sequence of different workload types arise elsewhere too.

load balancing for replicated data; whereby due to NP-completeness of the (query) assignment problem a greedy approach is proposed. However, this approach is not transparent for cost models and does not support any workload administration (e.g., aggregation) in contrast to our approach. Moreover, cost-based query optimization is only applicable for joins according to the authors.

7.2. Related Work on Self-Tuning

Since decades, query optimization is an important research field. However, cost-based optimization is a costly task, whereas it becomes difficult to handle complexity in the 1980s (e.g., workload is not known in advance; cf. Chapter 2). Therefore, Freytag, Graefe, DeWitt, and Sellis [Fre87, GD87, Sel88] lay the groundwork for most today's optimizers. They propose rule-based-optimization approaches to prune the solution space for cost-based optimization which are integrated in all DBMSs nowadays. Moreover, new query-processing approaches for several operation (e.g., join) lead to a new query-processor type – Volcano [Gra90, GM93, Gra94b, CG94] – which is transparent to specific implementation of operators. Most query processors are Volcano-like nowadays. Nevertheless, rule-based optimization and proof of rules is in research [Gra00, ENR09] to the present. In contrast to our approach, rule-based and cost-based query optimization consider certain architecture only.

Researchers reveal hardness of access-paths selection (e.g., indexes) for optimal query processing on changing workloads (cf. Chapter 2). Hence, automatic selection approaches are proposed to support query processing – self-tuning configurations. We argue, the idea of self-tuning is omnipresent for dynamic database optimization in which workloads are analyzed to react on specific events. Somehow all self-tuning approaches are related to our approach. Due to the amount of research in the field of self-tuning, we only give a brief overview to key aspects. Chaudhuri and Narasayya [CN07] and Bruno [Bru11] give a good overview to the progress in this research field as well as the need of further tasks (e.g., mixed workloads or hardness of workload prediction for multi-tenancy).

First ideas for self-tuning are proposed by Rozen and Shasha [RS91], Mönkeberg et al. [MZHW93], respectively Weikum et al. [WHM+93, WHMZ94]; whereby the latter propose the feedback-control loop – *observe, predict, and react* – which is derived from control systems in automation engineering. Another pioneering approach – WATCHMAN – is proposed by Scheuermann et al. [SSV96] that considers intelligent caching strategies for DWHs. The authors propose algorithms, which, compared to the existing configuration, estimate optimal cost savings of a new configuration. Scheuermann et al. do not restrict themselves to certain design structure rather they aim on a general solution. Similar algorithms are proposed by Caprara et al. [CFM95] even though the algorithms focus on index selection.

In process of time, several approaches are developed from basic self-tuning ideas. To mention are self-tuning for index structures (e.g., what-if analysis by Chaudhuri and Narasayya [CN97, CN98]), which are extended to autonomous index tuning (e.g.,

by Sattler et al. [SGS03, SSG04, SSG05] and by others [ACN00, LSSS07b, Lüb07]). An overview to (automatic) index selection and its complexity can be found in [CDN04]. A second aspect of interest for self-tuning are materialized views (ASTs) [CKPS95, SDJL96, BM00, ZCL+00, LGB08, LGB09] and their maintenance [MQM97], which are under investigation for years. Furthermore, Agrawal et al. [ACN00] describe necessity of AST merging for online recommendation and administration.

Later on, researchers recognize the mutual impact of design structures (e.g., by Agrawal et al. [ACK+04] or by Zilio et al. [ZZL+04]) as well as the idea of self-tuning passes to structures itself (e.g., by Graefe and Kuno [GK10] or Idreos et al. [IKM07b, HIKY12]) or to reduce necessary effort (e.g., size of configurations by Heeren et al. [HJP03]. However, we raise no claim to completeness.

In line with the presented ideas, our approach follows the same core idea – self-tuning. Therefore, we analyze workloads to give recommendations and react on changing requirements (e.g., selection or creation of access paths), and improve query processing at large – mostly for reduction of costs and query execution times. In comparison to the mentioned approaches, the unique characteristics of our approach are the architecture-cross-cutting analysis and recommendation for query processing (and physical design).

7.3. Related Work on Physical Design

In this section, we discuss advisors respectively alerter for physical design due to the fact; our approach proposes physical design with respect to storage architectures. Advisors are primarily based on workload decomposition and design-selection approaches (e.g., for indexes) from previous research, which we do not discuss separately. Researchers and companies target on automatic physical design because the design is crucial for DBS performance. Unfortunately, the selection problem is NP-complete for each design structure. Moreover, there exists interaction between design structures for query performance (e.g., indexes on materialized views).

Most prominent approaches are the DB2 Design Advisor [VZZ+00, ZRL+04, ZZL+04], the Database Tuning Advisor/Alerter by Microsoft [ACK+04, BC06, BC07], and the Oracle SQL Access Advisor [Ora03b] respectively the Oracle Tuning Advisor [DDD+04][4]. All approaches have in common that they analyze workloads to recommend sets of physical-design structures (i.e., indexes and materialized views) – namely configurations – that optimize data (path) access and query execution. Therefore, all advisors gather necessary statistics from optimizers respectively internal caches/catalogs which is a straightforward – but effective – idea and comparable to our approach.

However, Zilio et al. [ZZL+04] and Agrawal et al. [ACK+04] highlight necessity for joint considerations for indexes and materialized views to obtain most efficient design configuration. Both approaches also take partitioning into account; whereby

[4] We consider approaches of Microsoft and Oracle as one tool for ease of explanation.

Zilio et al. [ZRL+04] take additionally multi-dimensional clustering into account. Nevertheless, it is irrelevant how advanced these approaches are or will be, they are limited to one DBMS or at least to one architecture; whereas our approach focuses on physical design in terms of architecture selection.

We argue, advisor approaches for architecture-dependent physical design can be used in corresponding parts of hybrid systems. For architecture-dependent physical design, we assume limited tuning capabilities due to dynamic workload dispatching to different storage architectures, thus, even if a workload is known in advance, query-dispatching result will not be known before query evaluation. Nevertheless, a hierarchical optimization is conceivable on a coarse-grained level (e.g., for specific query types or sequences).

We are sure that there exist many other design-advisor approaches (and not for Row Stores only), which are not such prominent in literature. We believe sorted projections in C-Store [SAB+05, Aba08, ABH+13] respectively Vertica (i.e., sorted multi-column column-wise compressed projections) are candidates for automatic physical design. However, to the best of our knowledge other approaches are focused to specific DBMSs or architectures. Finally, we highlight that there exist related advisor approaches for other purposes (e.g., IBM DB2 Configuration Advisor [KLS+03]). The IBM Configuration Advisor recommends pre-configurations for DBSs.

7.4. Related Work on Relational Storage Architectures

The relational data model is not limited to certain architecture due to implementation transparency which is a key finding of Codd [Cod70]. Therefore, we discuss different architectures that implement the relational data model besides classic Row Stores like System R/DB2, INGRES, Oracle, SQL Server to name some.

New requirements induce a separation of systems to specialized DBSs for OLTP and OLAP in the 1990s. Later on, researchers recognize that even specialized systems (e.g., DWHs) encounter problems with even newer requirements (e.g., too long time delay by ETL). Researchers attend to such issues especially for OLAP systems (e.g., dimension updates by Vaisman et al. [VMRC04]). Others even go further for (near) real-time OLAP (e.g., real-time ETL [SB08], real-time update and query [ZAL08]).

Shorten update times on OLAP systems is not the only problem which researchers have to engage. Researchers recognize that specialized systems hit the wall due to the explosion of data amount (e.g., described by Korth and Silberschatz [KS97]). A well-known idea [CK85, CY90] advocates the development of a new architecture – Column Store – in which data-storage methodology (i.e., decomposition) is inherently suitable for OLAP. A pioneering approach by Sybase (i.e., Sybase IQ) exists since early 1990s. However, time has come for Column Stores in the 2000s, where several open- and closed-source Column Stores were introduced [SAB+05, ZBNH05, LLR06, Ing09, Aba08, SWES08]. Differences of historic approaches to state-of-the-art Column Stores are discussed in [HLAM06, ABH09].

Furthermore, Abadi et al. [AMH08] compare performance of Row Stores and Column Stores on a star-schema benchmark (i.e., SSDBM). They simulate Column Stores in a Row Stores by indexing each column or vertical partitioning of the whole schema. That is, usage of Column Store functionality in a Row Stores is possible but the performance is poor. We argue, Abadi et al. use a classic DWH benchmark that does not consider new requirements in the OLAP domain, which we discussed before. Nevertheless, such studies show that system separation becomes a separation of architectures. That is, Column Stores are more suitable for OLAP; whereas Row Stores are more suitable for OLTP. Moreover, either architecture cannot outperform the other in their inherent domains. Thus, we have to select a storage-architecture based on the domain. Some Column Store implementations are not designed for any (ad-hoc) DML (e.g., Infobright [SWES08, Inf08, Inf11a, Inf11b]) because they are designed for classic DWH environment with ETL. Others support DML but still try to cope with update problems known from basic approaches (e.g., by Copeland and Khoshafian [CK85]) already. Thus, systems need solutions that overcome (or at least mitigate) update-processing issues. Therefore, some systems introduce writable storages respectively nodes, in which DML statements take effect and subsequently, are merged into read-optimized (column-oriented) storage (e.g., C-Store/Vertica [SAB+05] or Sybase [Syb10, How10]). That is, such storages work like a mini ETL but in contrast, workload can access writable storages already. For a comprehensive overview to Column Store implementations, we refer to the book chapter by Abadi et al. [ABH+13].

We state that such storage separation for DML statements is not comparable to OLTP, thus, storage-architecture selection is still necessary and is even more crucial for mixed OLTP/OLAP workloads. Our approach is related to storage architectures in terms of selection of these with respect a given workload (domain). However, our approach is not related to architectures in terms of improvements which cope with architecture-specific drawbacks to either domain. We introduce our storage-architecture advisor which assists storage-architecture selection especially for mixed OLTP/OLAP workloads. We do not deny that approaches exist for mixed OLTP/OLAP workloads. As hardware gets cheaper, in-memory DBMSs become attractive. In recent years, in-memory DBMSs are developed (e.g., HyPer [KN10, KN11] or HANA [Pla09]) that focus on requirements for mixed OLTP/OLAP workloads. Still, we argue that even today, not each DBS is suitable to run in-memory (e.g., monetary or environmental constraints). We propose a more general approach assisted by our advisor approach that is also suitable for disk-based DBSs.

7.5. Related Work on Hybrid Database Management Systems

In this section, we discuss approaches which support (a) NSM and DSM respectively something in between or (b) OLTP beside their OLAP capability.

For sake of completeness, we mention Column Stores with writable storages [SAB+05, Syb10, How10]. Such systems are designed for OLAP applications and support DML-statement processing via special storages (i.e., instead of ETL), but not capable for OLTP by design. Even if writable storages are strict NSM-like, we do not refer to hybrid systems for such Column Stores. An important aspect is related to our approach – these systems are not main-memory centric. That is, most data does not have to be in-memory.

An approach which is hybrid by its design is called Partition Attributes Across (PAX) [ADHS01, Böß09]. In PAX, data is stored column-wise within a page (or a file) as for the DSM; but all columns of a tuple are kept on the same page instead of a page per column. PAX is an interesting approach but not applicable for our purpose. First, PAX is main-memory centric as it focuses on caching behavior; whereas it has no impact on I/O performance. Second, PAX needs extra effort for the complex page layout and inner-page administration (e.g., space management) which has issues with variable length attributes (e.g., recompaction on changing data). In contrast to PAX, our research focuses on hybrid solution that is suitable for disk-based DBSs as well.

The same statement holds for an approach by Zukowski et al. [ZNB08] implemented in MonetDB (i.e., it is main-memory centric). This approach supports NSM and DSM in one system which is able to convert from one representation to the other on-the-fly (i.e., in-memory). In line with Zukowski et al. [ZNB08], we observe benefits to convert from NSM to DSM and vice versa during query processing. In contrast, we do not only focus on trade-offs for CPU caused by tuple reconstructions.

Another approach by Schaffner et al. [SBKZ08] introduces a Column Store in a MaxDB-based OLTP system that holds a subset of OLTP data (e.g., in SAP Business Warehouse). That is, a build-in DWH is proposed based on TREX [LLR06] that supports build-in ETL via queue tables. Another in-memory approach, which is highly related, proposed by Plattner [Pla09] claims that an in-memory Column Store is suitable for OLTP and OLAP processing; whereas Plattner supposes an insert-only approach for update processing (i.e., like historicizing in DWHs). In [Pla11], ideas are substantiated with implementation of column groups, differential store for superior OLTP performance (i.e., as described above for other Column Stores), and aging methodology for data-volume reduction which needs to be hold in main memory. Aged data (i.e., passive/cold) is hold in a flash-based history store. However, both approaches have some influence to the later on featured HANA appliance; whereby, column groups are omitted from the original idea of Plattner. In HANA, both the Column Stores and the Row Stores are fully kept in-memory[5]; whereby persistence is achieved by savepoints and logs[6] on high-performance storage [Wor12]. Plattner states in [Pla11] himself that the proposed approach does not fit on a single server blade, thus, it runs on a cluster of blades (e.g., 2TB main memory, 64 CPU cores). Consequently, the HANA approach is outstanding but not feasible for our purpose.

[5] Architecture overview of the HANA system: `http://saphanatutorial.com/an-insight-into-sap-hana-architecture/`.

[6] Persistent storage for backup and recovery in HANA: `http://saphanatutorial.com/sap-hana-backup-and-recovery/`.

We note that Sherkat et al. [SFA+16] also recognize problems with large resource requirements by HANA. Therefore, they introduce piecewise columnar access for reduction of the data amount which needs to be hold in-memory. That is, value identifier vectors – representing data portions – are split into chunks (i.e., 64-bit words) of paged data vectors. Nevertheless, main aspects of HANA and its resource consumption remain, thus, HANA scales up well for high performance applications, but it is not suitable for smaller applications. In contrast, our approach shall scale down to smaller disk-based DBMSs as well. A summary for the architecture of HANA and more detailed considerations on complete appliance can be found in [Wor12].

However, another project – namely HYRISE [GKP+10] – initiated by Plattner is closely related to the ideas of Schaffner et al. [SBKZ08] and Plattner himself [Pla11]. In contrast to HANA, column groups are of major matter. In other words, HYRISE supports vertical partitioning of relational tables into arbitrary-wide disjoint sets of attributes [GKP+10]. That is, the storage manager decides how data is stored – row-wise, column-wise, or somehow in between. Like HANA, HYRISE divides the storage into read-optimized main store and write-optimized differential store (i.e., delta store) [KKG+11]. Subsequently, Wust et al. [WGP13] introduce priority settings for queries; whereby a query is translated to a set of tasks. Tasks are atomic of varying size and can be independently executed [WGH+14]. We state, we are not sure about utilization of the storage manager for the ongoing project due to the fact that primarily subsequent research refers to the Column Store only (i.e., with main and delta store; e.g., in [KKG+11, WGH+14]). In line with [AIA14], we argue that the storage manager adopts ideas of data morphing [HP03] – a cache-miss-cost model – and proposes (optimal) vertical partitioning concerning cache performance for a given workload. Nevertheless, a combination of our storage manager (for adaptive storage) with priority settings is an important aspect for future directions of our approach (cf. Section 8.3); but like HANA, HYRISE has immense resource requirements especially main memory. Summarizing, HYRISE is in contrast to our approach main-memory centric and does not support adaptive storage.

An approach by Kemper and Neumann (e.g., [KN10, KN11]) goes further and solely focuses on optimal in-memory processing (i.e., excluding archive on external high-performance-storage server). The HyPer system is designed for OLTP and close to real-time OLAP; whereby HyPer can be Row Stores or Column Store (i.e., configurable). That is, a HyPer instance uses the same architecture for OLAP and (serialized) OLTP. Therefore, HyPer strips away many comfortable and transparent approaches (e.g., buffer management) to achieve maximum performance via utilization of modern hardware and OS tools. However, the performance of HyPer is more than competitive and scales with corresponding hardware (i.e., more main memory for multiple forks), but a higher investment on hardware is needed for a general purpose in-memory DBS instead of specialized separated (disk-based) systems. That is, high-level commodity hardware is needed for an initial relative small benchmark (i.e., 12 warehouses; cf. [KN11, FKN12]); in comparison, we use same benchmark with 90 warehouses resulting in approximate 10GB initial size of raw data. We argue that main-memory centric respectively in-memory DBMSs scale up with outstanding

results, but they come to sweat whenever system has to scale down (e.g., data does not fit in main memory). We investigate on the trade-off between OLTP/OLAP support in one DBS and disk-based DBMSs.

We state, several additional powerful OLAP add-ons by DBMS vendors exist. To name but a few, Microsoft Apollo [Han10, LBH+15] – at first, read-only, updatable in-memory add-on nowadays, IBM IDAA [MKKI13] – OLAP-server add-on for high performance server (e.g., mainframe with InfiniBand), and DB2BLU [RAB+13] – in-memory for read-mostly OLAP. These approaches aim at in-memory high performance analysis but not at hybrid workloads, thus, they leave the focus of our work.

An approach by Alagiannis et al. – which is designed for analytical queries – does not hold data redundantly but adapts the internal storage layout in advance to query execution on table level [AIA14]. This approach – H_2O – shows better results on (read-only) workloads than their corresponding Column Store and Row Store implementations. Furthermore, the cost model uses CPU and I/O costs for estimation of query-execution cost as we have suggested, but we are not limited to. However, they reorganize storage layout via early materialization whenever query benefits from the new storage layout. What in our opinion is left open how storage layout is written back to disk – they only state, data may be read from disk, thus, the approach is main-memory centric in our opinion. Furthermore, the approach uses a (static or dynamic) window (i.e., number of queries), where other self-tuning approaches use thresholds to solve problems with inflated reorganization cost respectively with oscillating workloads. Alagiannis et al. state that the dynamic window adopts on workload changes but leave open how this is achieved. However, the work on on-the-fly created operators is very path-breaking for hybrid query processing and thus, we emphasize deeper inspection for integration into our approach. We argue that adaptive in-memory storage-layout is more smoothly and cheaper than on disk, thus, we can consider different granularities for adaptive storage layout in main memory and on disk. However, one of our core aspects – the OLTP support – is not considered, thus, we cannot figure out the impact of OLTP to this approach.

The approach of Arulraj et al. [APM16] extends ideas of H_2O in several ways. First, the flexible storage model (FSM) adds horizontal partitioning (i.e., number of tuples in a partition by parameter); whereby vertical and horizontal partitioning is not limited to the table level. Therefore, they introduce physical and logical tile; whereas physical tiles hold data in a flexible schema and logical tiles hold offsets of tuples spread over several physical tiles (i.e., comparable to indexes). In contrast to pages, a reference is prohibited between logical tiles, which may induce recomputation of logical tiles. Second, Arulraj et al. reconsider combination of query execution and data reorganization. That is, they hide specific storage layouts from query-execution engine; whereby storage layout is incrementally reorganized in background (in contrast to H_2O). For the partitioning, they use a clustering approach to identify attributes which are accessed together and a greedy algorithm to generate the (tile) storage layout. Arulraj et al. present significant ideas (i.e., which we can adopt), but we identify also drawbacks. The insert-only approach (via append

Approach	OLTP Support	Adaptive	Multiple Stores	Main-memory Centric
Column Stores with write storage[7]	○	○	○	○
PAX [ADHS01] / SPAX [Bößß09]	●	○	○	●
Zukowski et al. [ZNB08] (MonetDB)	–	○	●	●
SAP BW/HANA [SBKZ08, Pla09, Pla11, SFA+16]	●	○	●	●
HYRISE [GKP+10, KKG+11, WGP13, WGH+14]	●	○	●	●
HyPer [KN10, KN11, FKN12]	●	○	●	●
H_2O [AIA14]	○	●	●	●
FSM [APM16]	●	●	●	●
AQUA^2	●	●	●	○

Table 7.2.: *Comparison of key aspects for workload decomposition and representation. Legend: ● fulfilled, ○ not fulfilled, – no information available.*

like HANA) demolishes data locality if not reorganized. Furthermore, workloads are serialized used for experiments (i.e., first analytical query then transactions), thus, transactions have an impact on overall system load but not on processing of analytical queries due to MVCC. That is, we argue that Arulraj et al. use read-mostly OLAP-like workloads for main evaluation. Arulraj et al. present more write-heavy workload in supplementary material that is still based on their own benchmark composed of an insert and a (simple) select with varying projectivity and selectivity. We argue, neither the insert statement represents OLTP nor the query represents OLAP sufficiently. Moreover, their approach is again main-memory centric (i.e., an in-memory DBMS). Finally, FSM possesses most core aspects of our work (i.e., for mixed OLTP/OLAP with reservations) and ideas for integration in our approach but lacks of disk-based DBS support (i.e., capability to scale down).

We summarize our considerations on hybrid DBMSs in Table 7.2. Column Stores do not comply with key aspects of our work as anticipated; whereas related approaches comply with OLTP (PAX) or multiple store support (MonetDB), but are not adaptive and are main memory-centric in contrast to our approach. Other approaches (HANA, HYRISE, HyPer) support OLTP and multiple stores, but likewise they are not adaptive storage and are main memory-centric. Only two approaches (H_2O, FSM) are very closely related to our approach and comply with most key aspects (i.e., OLTP support with reservations). However, they are still main-memory centric and do not scale down well, which is a major aspect of our work and our motivation from the outset.

[7] We summarize Column Stores to a group for conservation of clear arrangement due to the fact that basic idea is the same (e.g., [SAB+05, Syb10, How10]).

8. Conclusion

Physical design is beneficial, complex, and crucial for Database Systems (DBSs) throughout their lifetime. Originally, such design was a manual task, in which administrators create index structures for a faster data access. The amount of research on index structures and index selection shows the significance of this problem over decades. Besides the number of different index structures, which makes decision for an index type difficult, the selection of index sets is even more difficult (i.e., the index-selection problem is NP-complete) for given constraints (e.g., storage space). Thereby, tools assist database designers figuring out near-optimal index configurations (cf. Chapter 2). Over decades new approaches and new requirements advocate more and more innovative approaches which increase solution space for physical design. Column-oriented Database Management Systems (Column Stores) were one approach for relational Database Management Systems (DBMSs) that store data column-wise, in contrast to row-wise storage in Row-oriented Database Management Systems (Row Stores). Column Stores involve several advantages especially for analytical workloads, but raise a dilemma for physical design where certain architecture has to be selected as precedent stage for physical design DBSs (cf. Chapter 3). In contrast to provided tools for physical design of DBSs[1], tool support for storage-architecture selection is an open problem.

In this thesis, we introduced a new approach that extends ideas of existing tool sets for physical design of relational DBMSs (e.g., advisor for index configurations) to assist database designers with tool-supported selection of storage architectures. That is, our approach to select the optimal storage architecture integrates well with existing approaches which allow database designers the tuning of physical design for a certain DBSs. In consequence, designers can select the optimal storage architecture at first and at second select optimal DBS-specific design. For tool-supported design decisions, they do not need expert knowledge as they would need for manual system design (i.e., architecture or DBMS specifics). Design tools allow users to decide for the optimal architecture with a given workload (sample) and DBSs (i.e., their statistics). Our approach ensures the optimal selection in consideration of uncertainty of input variables (e.g., quality of samples) even if not all cost measures are available. However, we do not affect usability of existing design tools whenever these exist for the selection of physical design and (optimal) storage architecture, respectively. From now on, designers are able to verify with our approach, if the architecture selection is suitable for the given application, before invest in misdirected physical design tuning on less-than-ideal architecture (i.e., DBMS). Subsequently,

[1] Most physical design tools are limited to a certain DBMS.

we proposed ideas for hybrid (relational) DBMSs and DBSs which are intended to cope with challenges for mixed requirements and are not satisfactorily resolved by existing architectures. That is, hybrid systems should overcome drawbacks of other architectures by a suitable and sophisticated trade-off for mixed requirements. We implemented our composed ideas – which we call Automated Query Interface for Relational Architectures (AQUA$^{\mathbf{2}}$) – in a prototypical hybrid DBMS as proof of concept. Afterwards, we generalized our approach in an intellectual game (i.e., potential future work).

8.1. Thesis Summary

In Chapter 2, we discussed general concepts of the relational data model, relational storage architectures, different application fields for relational DBMSs, and (potential) optimization methods. Thereby, we introduced the background for our work and gave assistance for understanding and classification of our work for readers who were unfamiliar with relational data-management concepts.

In Chapter 3, we surveyed two storage architectures for the relational data model – Row Store and Column Store – and discussed their benefits and drawbacks. The survey showed necessity for storage-architecture selection as well as new challenges for either approach on mixed requirements for former disjoint application domains. In this context, we presented a brief study for a certain application domain on both architectures that substantiated (design) challenges by ambiguous results.

In Chapter 4 and Chapter 5, we proposed and evaluated our storage-architecture-selection approach. Therefore, we introduced an architecture-independent workload-decomposition and -representation approach in Chapter 4 which enabled us to store simultaneously statistics of different systems. In Chapter 5, we discussed cost estimation based on our workload representation that both were combined and evaluated in terms of a storage advisor. Moreover, we discussed three different decision situations which can be resolved with our approach.

In Chapter 6, we presented concepts for hybrid relational storage architectures. We introduced our query-interface framework, discussed different optimization methods, and presented heuristics for hybrid systems. Subsequently, we showed results of a mixed (requirement) workload on two different hybrid system setups – two replicated DBMSs and our prototypical implementation of a hybrid DBMS.

In Chapter 7, we discussed related research in comparison to the key aspects of our approach. Therefore, we focused on the following five key aspects: workload representation, self-tuning, physical design, relational storage architectures, and hybrid DBMSs. We especially discussed hybrid DBMSs in more detail.

8.2. Contribution

As major aspect, we extended physical-design process with respect to decisions on storage architectures for given workloads. Therefore, we provided an architecture-

selection (and -verification) approach which is executed before one brings existing design approaches into action. We analyzed state-of-the-art approaches for relational data management – Row Store and Column Store – with respect to their capabilities for a certain workload domain. Furthermore, we advocate for hybrid architectures, as benefits became obvious for a mix of classic disjoint workload domains. We proposed, implemented, and evaluated a tool-supported storage-architecture selection which we name at large AQUA$^{\mathbf{2}}$. In detail, we contributed the following four aspects:

1. We introduced an approach for architecture-independent workload decomposition, representation, and normalization to compare storage architecture inplane. We derived our workload-pattern approach straightforwardly from internal query representation in relational DBMSs in which statistics are stored from query optimizers respectively user samples with respect to architecture specifics. We showed necessity of statistics normalization due to vendor-specific and architecture-specific internal different representation. We integrated the workload-representation approach into the following aspects of our work.

2. We proposed our storage-architecture-selection approach for given workloads based on workload patterns. We showed the arbitrary degree of detail of architecture selection (i.e., in dependency of detail degree of statistics); whereas the transparency for cost functions and thus, for cost criteria has been shown. Despite the statistic-based architecture selection, we proposed selection approaches for unknown workloads considering uncertainty and multi-criteria decision with user weights. Moreover, we derived heuristics for physical design from our experiments. We highlighted that the storage-architecture selection is applicable for DBS redesign or a-priori design on known workloads and user samples. Our selection approach was not directly pertinent to hybrid architectures; nevertheless, this work was crucial for further research (i.e., query processing in hybrid storage systems).

3. We composed our workload-representation and architecture-selection approaches as hybrid query interface (AQUA$^{\mathbf{2}}$). We analyzed queries (i.e., parts of a workload) within our query interface and decided where queries are executed best on hybrid stores which support both architectures redundantly. Therefore, we proposed stepwise optimization – rule-based on the global level and cost-based on the local (storage) level. For the rule-based optimization, we have proposed query-execution heuristics which we implemented in our prototypical hybrid store.

4. We developed a hybrid DBMS with two architectures and query engines. Therefore, we implemented additional Column Store functionality in an open-source Row Store. Queries are rule-based dispatched in the hybrid DBMS prototype by the integrated AQUA$^{\mathbf{2}}$ framework. Subsequently, we evaluated our prototypical implementation with AQUA$^{\mathbf{2}}$ integration based on a mixed domain benchmark; whereas we could show significant improvements in com-

parison to the Row Store implementation. Finally, we outlined potential improvements to our prototype and ideas for (a more-general) hybrid DBMS implementation.

8.3. Future Work

We have suggested several improvements to (a) storage-architecture selection and (b) hybrid stores. However, we observe several improvements remain for consideration which we left open with our approach. We believe that a first improvement is a further automation of the architecture-selection process, as we have to provide an extraction mechanism for each DBMS and therefore, we need knowledge of internal query-plan representation. We argue, a limitation to interface-based extraction (e.g., JDBC) may solve the extraction issue even though a number of DBMSs will be excluded from analysis. To tackle the problem of internal query-plan representation, text-recognition respectively text-processing approaches may be suitable for mapping of operations to workload patterns. Such approach also allows further analysis on commonness and concurrency of workload parts (e.g., cf. Favre et al. [FBB07]).

We did not use automatic analysis and generation or rules for our set of heuristics. Machine learning may be another direction of research to increase the number of heuristics and improve their soundness. Therefore, a large enough number of different DBMSs for both architectures has to be analyzed on suitable benchmark(s), thus, analysis of statistics via machine learning results in sound rules. Our architecture-independent workload-representation approach can be utilized to provide necessary data for learning and test sets, whereas the decision approach would be enriched by machine learning methods. Sound design rules would ease basic physical design. Integration into AQUA^2 is promising for solution-space pruning as dispatching in hybrid DBSs would be improved and calculation costs for adaptive storage would be lighten in hybrid DBMSs.

We did not provide cost-based optimization in our prototypical hybrid store (cf. Section 6.4.3). In fact, implementation of cost-based query optimizer is necessary for both query engines as we advocate optimal support of our storage-architecture-decision approach. Cost-based optimization would also enable design-alerter functionality within hybrid stores that only needs extension of cost functions with modification cost for redesign. Henceforth, redesign and thus, adaptive storage allows redundancy reduction for hybrid stores as full redundancy is the major drawback of our implementation (e.g., higher effort for consistency or higher storage-space consumption). The ideal way would be without redundancy when data is stored only once in either architecture respectively a representation in between.

We believe that two suitable ways exist for redundancy reduction in hybrid stores which support both architectures redundantly (e.g., our prototype). First, implementation and integration of hybrid storage approaches, as we discussed in Section 7.5 (e.g., H_2O [AIA14], FSM [APM16]), into disk-based hybrid stores, whereas we should focus on adaptive approaches which use redundancy-free data representations. Due

to main-memory centric behavior of such approaches, impacts to disk-based DBSs – as we aim at – is currently left open. We advocate more research on this aspect of hybrid stores. However, such hybrid approaches make a query engine necessary which is able to process on Row Store, Column Store, and everything in between. That is, complexity increases due to potential query rewriting as it is known from distributed DBMS (cf. also Section 6.1.2).

Second, implementation of aging algorithms in which redundant data exists only during asynchronous physical reorganization. A promising approach was proposed by Funke [FKN12, Fun15] which clusters data into hot (i.e., volatile) and cold parts (i.e., rarely modified). That is, hot data is uncompressed on small (memory) pages as needed for optimal OLTP support; whereas cold data is compressed on huge (memory) pages). In between hot and cold data representation, Funke proposed transitional stages which implement the cooling process for data (i.e., sliding window to transfer data from hot into cold part and vice versa). We believe that such approach is not only suitable for compressed versus uncompressed data representation on horizontally partitioned Column Store, but also for adaptation of the storage architecture in a hybrid store based on a sophisticated age-limit. In consequence, the comparison of both implementations is of research interest.

Despite improvements for hybrid DBMSs, we believe that processing with our interface on hybrid DBSs would be improved via load-balancing, time-bound partitioning, and intra-query parallelization (i.e., merge of intermediate results needed). Furthermore, approaches for reusability in query processing may be interesting (e.g., for query plans by Ghosh et al. [GPSH02]) especially in combination with approaches for workload evolution, which imply statistics recalculation (e.g., by Favre et al. [FBB07]) and thus, imply query-plan recalculation in a combined consideration. In summary, we state that various interesting research topics arise from the combined point of view on design aspects and decision making as well as from weak spots which want to be fixed in future.

A. Appendix

Appendix A shares material with [LGB08, LGB09, Lüb09, LS10, Lüb10, LKS10, LKS11a, LKS11c, LKS11b, LKS11c, LKS12, WKLS12, LSKS12, LSS13].

In this chapter, we present additional information and results concerning our framework. That is, we restrict ourselves to sufficient large examples to improve the readability in previous chapters. Furthermore, we give extra information for interested readers to improve the comprehension and to support the replicability of our approaches.

A.1. Our Query Set for TPC-H 2.8

In the following, we present the remaining queries from the TPC-H Benchmark that we used for first considerations and our case study in Chapter 3.

```
1 SELECT s_acctbal,s_name,n_name,p_partkey,p_mfgr,s_address,s_phone,s_comment
2 FROM part,supplier,partsupp,nation,region
3 WHERE p_partkey = ps_partkey AND s_suppkey = ps_suppkey AND p_size = 25 AND p_type LIKE
    '%NICKEL' AND s_nationkey = n_nationkey AND n_regionkey = r_regionkey AND r_name =
    'AMERICA' AND ps_supplycost = (
4     SELECT MIN(ps_supplycost)
5     FROM partsupp,supplier,nation,region
6     WHERE p_partkey = ps_partkey AND s_suppkey = ps_suppkey AND s_nationkey = n_nationkey
        AND n_regionkey = r_regionkey AND r_name = 'AMERICA')
7 ORDER BY s_acctbal DESC,n_name,s_name,p_partkey
8 LIMIT 100;
```

Figure A.1.: *TPC-H query Q2 [Tra08].*

```
1 SELECT l_orderkey,SUM(l_extendedprice * (1 - l_discount)) AS revenue,o_orderdate,o_shippriority
2 FROM customer,orders,lineitem
3 WHERE c_mktsegment = 'FURNITURE' AND c_custkey = o_custkey AND l_orderkey = o_orderkey
    AND o_orderdate < date '1995-03-10' AND l_shipdate > date '1995-03-10'
4 GROUP BY l_orderkey,o_orderdate,o_shippriority ORDER BY revenue desc,o_orderdate LIMIT 10;
```

Figure A.2.: *TPC-H query Q3 [Tra08].*

```
1 SELECT o_orderpriority,COUNT(*) AS order_count
2 FROM orders
3 WHERE o_orderdate >= date '1994-07-01' AND o_orderdate < date '1994-07-01' + interval '3'
    month AND EXISTS (
4     SELECT * FROM lineitem WHERE l_orderkey = o_orderkey AND l_commitdate < l_receiptdate)
5 GROUP BY o_orderpriority ORDER BY o_orderpriority;
```

Figure A.3.: *TPC-H query Q4 [Tra08].*

```
1 SELECT supp_nation,cust_nation,l_year,sum(volume) AS revenue
2 FROM (SELECT n1.n_name AS supp_nation,n2.n_name AS cust_nation,EXTRACT(year FROM
    l_shipdate) AS l_year,l_extendedprice * (1 - l_discount) AS volume
3     FROM supplier,lineitem,orders,customer,nation n1,nation n2
4     WHERE s_suppkey = l_suppkey AND o_orderkey = l_orderkey AND c_custkey = o_custkey AND
        s_nationkey = n1.n_nationkey AND c_nationkey = n2.n_nationkey AND (
5         (n1.n_name = 'EGYPT' AND n2.n_name = 'INDONESIA') OR (n1.n_name = 'INDONESIA'
            AND n2.n_name = 'EGYPT'))
6     AND l_shipdate between date '1995-01-01' AND date '1996-12-31') AS shipping
7 GROUP BY supp_nation,cust_nation,l_year ORDER BY supp_nation,cust_nation,l_year;
```

Figure A.4.: *TPC-H query Q7 [Tra08].*

```
1 SELECT o_year,SUM(CASE WHEN nation = 'INDONESIA' THEN volume ELSE 0 END) / SUM(volume)
    AS mkt_share
2 FROM (SELECT EXTRACT(year FROM o_orderdate) AS o_year,l_extendedprice * (1 - l_discount) AS
    volume,n2.n_name as nation
3     FROM part,supplier,lineitem,orders,customer,nation n1,nation n2,region
4     WHERE p_partkey = l_partkey AND s_suppkey = l_suppkey AND l_orderkey = o_orderkey AND
        o_custkey = c_custkey AND c_nationkey = n1.n_nationkey AND n1.n_regionkey = r_regionkey
        AND r_name = 'ASIA' AND s_nationkey = n2.n_nationkey AND o_orderdate BETWEEN date
        '1995-01-01' AND date '1996-12-31' AND p_type = 'LARGE BRUSHED COPPER') AS
        all_nations
5 GROUP BY o_year ORDER BY o_year;
```

Figure A.5.: *TPC-H query Q8 [Tra08].*

```
1 SELECT nation,o_year,SUM(amount) AS sum_profit
2 FROM (SELECT n_name AS nation,EXTRACT(year FROM o_orderdate) AS o_year,l_extendedprice * (1
    - l_discount) - ps_supplycost * l_quantity AS amount
3     FROM part,supplier,lineitem,partsupp,orders,nation
4     WHERE s_suppkey = l_suppkey AND ps_suppkey = l_suppkey AND ps_partkey = l_partkey AND
        p_partkey = l_partkey AND o_orderkey = l_orderkey AND s_nationkey = n_nationkey AND
        p_name LIKE '%papaya%') AS profit
5 GROUP BY nation,o_year ORDER BY nation,o_year desc;
```

Figure A.6.: *TPC-H query Q9 [Tra08].*

```
1 SELECT c_custkey,c_name,SUM(l_extendedprice * (1 - l_discount)) AS
    revenue,c_acctbal,n_name,c_address,c_phone,c_comment
2 FROM customer,orders,lineitem,nation
3 WHERE c_custkey = o_custkey AND l_orderkey = o_orderkey AND o_orderdate >= date '1994-12-01'
    AND o_orderdate < date '1994-12-01' + interval '3' month AND l_returnflag = 'R' AND c_nationkey
    = n_nationkey
4 GROUP BY c_custkey,c_name,c_acctbal,c_phone,n_name,c_address,c_comment ORDER BY revenue
    DESC LIMIT 20;
```

Figure A.7.: *TPC-H query Q10 [Tra08].*

```
1 SELECT ps_partkey,SUM(ps_supplycost * ps_availqty) AS value
2 FROM partsupp,supplier,nation
3 WHERE ps_suppkey = s_suppkey AND s_nationkey = n_nationkey AND n_name = 'SAUDI ARABIA'
4 GROUP BY ps_partkey HAVING SUM(ps_supplycost * ps_availqty) >
5     (SELECT SUM(ps_supplycost * ps_availqty) * 0.0001000000
6     FROM partsupp,supplier,nation
7     WHERE ps_suppkey = s_suppkey AND s_nationkey = n_nationkey AND n_name = 'SAUDI
        ARABIA')
8 ORDER BY value DESC;
```

Figure A.8.: *TPC-H query Q11 [Tra08].*

```
1 SELECT l_shipmode,SUM(CASE WHEN o_orderpriority = '1-URGENT' OR o_orderpriority = '2-HIGH'
    THEN 1 ELSE 0 END) AS high_line_count,SUM(CASE WHEN o_orderpriority <> '1-URGENT' AND
    o_orderpriority <> '2-HIGH' THEN 1 ELSE 0 END) AS low_line_count
2 FROM orders,lineitem
3 WHERE o_orderkey = l_orderkey AND l_shipmode IN ('REG AIR', 'MAIL') AND l_commitdate <
    l_receiptdate AND l_shipdate < l_commitdate AND l_receiptdate >= date '1993-01-01' AND
    l_receiptdate < date '1993-01-01' + interval '1' year
4 GROUP BY l_shipmode ORDER BY l_shipmode;
```

Figure A.9.: *TPC-H query Q12 [Tra08].*

```
1 SELECT 100.00 * SUM(CASE WHEN p_type LIKE 'PROMO%' THEN l_extendedprice * (1 -
    l_discount) ELSE 0 END) / SUM(l_extendedprice * (1 - l_discount)) AS promo_revenue
2 FROM lineitem,part
3 WHERE l_partkey = p_partkey AND l_shipdate >= date '1996-05-01' AND l_shipdate < date
    '1996-05-01' + interval '1' month;
```

Figure A.10.: *TPC-H query Q14 [Tra08].*

```
1 SELECT SUM(l_extendedprice) / 7.0 AS avg_yearly
2 FROM lineitem,part
3 WHERE p_partkey = l_partkey AND p_brand = 'Brand#12' AND p_container = 'MED BOX' AND
    l_quantity < (
4     SELECT 0.2 * AVG(l_quantity) FROM lineitem WHERE l_partkey = p_partkey);
```

Figure A.11.: *TPC-H query Q17 [Tra08].*

```
1  SELECT SUM(l_extendedprice* (1 - l_discount)) AS revenue
2  FROM lineitem,part
3  WHERE (
4      p_partkey = l_partkey AND p_brand = 'Brand#31' AND p_container in ('SM CASE', 'SM BOX', 'SM
         PACK', 'SM PKG') AND l_quantity >= 6 AND l_quantity <= 6 + 10 AND p_size BETWEEN 1
         AND 5 AND l_shipmode in ('AIR', 'AIR REG') AND l_shipinstruct = 'DELIVER IN PERSON')
5  OR (
6      p_partkey = l_partkey AND p_brand = 'Brand#15' AND p_container IN ('MED BAG', 'MED BOX',
         'MED PKG', 'MED PACK')
7      AND l_quantity >= 18 AND l_quantity <= 18 + 10 AND p_size BETWEEN 1 AND 10 AND
         l_shipmode IN ('AIR', 'AIR REG') AND l_shipinstruct = 'DELIVER IN PERSON')
8  OR (
9      p_partkey = l_partkey AND p_brand = 'Brand#21' AND p_container IN ('LG CASE', 'LG BOX', 'LG
         PACK', 'LG PKG') AND l_quantity >= 27 AND l_quantity <= 27 + 10 AND p_size BETWEEN 1
         AND 15 AND l_shipmode IN ('AIR', 'AIR REG') AND l_shipinstruct = 'DELIVER IN PERSON');
```

Figure A.12.: *TPC-H query Q19 [Tra08].*

```
1  SELECT s_name,s_address
2  FROM supplier,nation
3  WHERE s_suppkey IN (
4      SELECT ps_suppkey FROM partsupp WHERE ps_partkey IN (
5          SELECT p_partkey FROM part WHERE p_name LIKE 'black%')
6      AND ps_availqty > (
7          SELECT 0.5 * SUM(l_quantity) FROM lineitem
8          WHERE l_partkey = ps_partkey AND l_suppkey = ps_suppkey AND l_shipdate >= date
             '1993-01-01' AND l_shipdate < date '1993-01-01' + interval '1' year))
9  AND s_nationkey = n_nationkey AND n_name = 'RUSSIA'
10 ORDER BY s_name;
```

Figure A.13.: *TPC-H query Q20 [Tra08].*

```
1  SELECT cntrycode,COUNT(*) AS numcust,SUM(c_acctbal) AS totacctbal
2  FROM (SELECT SUBSTRING(c_phone FROM 1 FOR 2) AS cntrycode,c_acctbal
3      FROM customer
4      WHERE SUBSTRING(c_phone FROM 1 FOR 2) IN ('18', '20', '27', '19', '25', '26', '33') AND
         c_acctbal > (
5          SELECT AVG(c_acctbal) FROM customer
6          WHERE c_acctbal > 0.00 AND SUBSTRING(c_phone FROM 1 FOR 2) IN ('18', '20', '27', '19',
             '25', '26', '33'))
7      AND NOT EXISTS (SELECT * FROM orders WHERE o_custkey = c_custkey)
8      ) AS custsale
9  GROUP BY cntrycode ORDER BY cntrycode;
```

Figure A.14.: *TPC-H query Q22 [Tra08].*

A.2. Query-wise Summary of I/O Costs for TPC-H 2.11.0

In the following tables (Table A.1 to A.17), we present our I/O-consumption results for queries of the TPC-H benchmark [Tra10] (version 2.11.0, scale factor 1) that were not discussed in Chapters 4 and 5 in more detail. We present our results query-wise for both systems (Oracle vs. ICE). The value in each pattern represents the summation of the corresponding operation costs (i.e., aggregated on sub-pattern level; cf. Section 4.3).

Workload	**Query Q1**			
Pattern	**Oracle** (22.82sec)		**ICE** (25sec)	
	Rows	I/O Costs	Rows	I/O Costs
Data Access	5789.7K	156321.522	6012.7K	5980
Group By	5789.7K	156321.5K	5916.6K	5980
Projection	5	0.135	4	65

Table A.1.: *Accessed data of TPC-H query Q1 - Number of rows and I/O costs in KBytes.*

Workload	**Query Q2**			
Pattern	**Oracle** (8.14)		**ICE** (41sec)	
	Rows	I/O Costs	Rows	I/O Costs
Data Access	10.2K	1452.133	1048.6K	1040
Non-vector	12K	1759.005	4521.9K	4455
Tuple Reconstruction			162.3K	195
Sort	316	78.526	460	65
Count	158	31.284	460	65
Projection	416	98.326	100	65

Table A.2.: *Accessed data of TPC-H query Q2 - Number of rows and I/O costs in KBytes.*

Workload	**Query Q3**			
Pattern	**Oracle** (30.97sec)		**ICE** (3sec)	
	Rows	I/O Costs	Rows	I/O Costs
Data Access	3984.7K	89977.195	7712K	7670
Non-vector	4204.7K	98117.269	9240.6K	9165
Tuple Reconstruction			177.6K	195
Group By	501.7K	30102.72	30.5K	65
Sort	501.7K	30102.72	11.6K	65
Count	501.7K	24082.172	11.6K	65
Projection	10	0.48	10	65

Table A.3.: *Accessed data of TPC-H query Q3 - Number of rows and I/O costs in KBytes.*

Workload	**Query Q5**			
Pattern	**Oracle** (32.66sec)		**ICE** (4sec)	
	Rows	I/O Costs	Rows	I/O Costs
Data Access	6389.5K	119631.12	7908.1K	7865
Non-vector	7777.1K	243824.536	14378.3K	14300
Tuple Reconstruction			1392.5K	1430
Group By	7.4K	844.854	7243	5980
Sort	50	3.65	5	65
Projection	25	2.85	5	65

Table A.4.: *Accessed data of TPC-H query Q5 - Number of rows and I/O costs in KBytes.*

Workload	**Query Q7**			
Pattern	**Oracle** (29.48sec)		**ICE** (4sec)	
	Rows	I/O Costs	Rows	I/O Costs
Data Access	3249.8K	63543.499	7908.1K	7865
Non-vector	3678.7K	102714.005	15685.4K	15600
Tuple Reconstruction			443.8K	455
Filtering			220.8K	260
Group By	5.6k	617.493	5.9K	65
Projection	1.5K	152.504	4	65

Table A.5.: *Accessed data of TPC-H query Q7 - Number of rows and I/O costs in KBytes.*

Workload	**Query Q8**			
Pattern	**Oracle** (29.95sec)		**ICE** (3sec)	
	Rows	I/O Costs	Rows	I/O Costs
Data Access	6619.1K	154010.467	8234.9K	8190
Non-vector	6700.3K	159438.031	22090.3K	21970
Tuple Reconstruction			494K	520
Group By	2446	364.454	2603	65
Projection	732	109.068	2	65

Table A.6.: *Accessed data of TPC-H query Q8 - Number of rows and I/O costs in KBytes.*

Workload	**Query Q9**			
Pattern	**Oracle** (37.05sec)		**ICE** (7sec)	
	Rows	I/O Costs	Rows	I/O Costs
Data Access	7511.2K	183432.882	8757.7K	8710
Non-vector	9212.7K	277307.627	33723.7K	33540
Tuple Reconstruction			1743.8K	1755
Group By	297.1K	38924.947	348.8K	390
Projection	42.5K	5571.823	175	65

Table A.7.: *Accessed data of TPC-H query Q9 - Number of rows and I/O costs in KBytes.*

Workload	**Query Q10**			
Pattern	**Oracle** (29.18sec)		**ICE** (10sec)	
	Rows	I/O Costs	Rows	I/O Costs
Data Access	2305.6K	75690.021	7973.4K	7930
Non-vector	2402.9K	79190.306	9476.6K	9425
Group By	97.21K	20900.15	114.7K	130
Sort	97.21K	20900.15	37.9K	65
Count	97.21K	17400.59	37.9K	65
Projection	97.23K	20903.73	20	65

Table A.8.: *Accessed data of TPC-H query Q10 - Number of rows and I/O costs in KBytes.*

Workload	**Query Q11**			
Pattern	**Oracle** (5.06sec)		**ICE** (1sec)	
	Rows	I/O Costs	Rows	I/O Costs
Data Access	810K	14470.029	1960.7K	1950
Non-vector	1610K	52070.029	2091.4K	2080
Tuple Reconstruction			64.2K	65
Group By	32K	1728	31.7K	65
Sort	832K	15648	29.8K	65
Projection	64K	2496	752	65

Table A.9.: *Accessed data of TPC-H query Q11 - Number of rows and I/O costs in KBytes.*

Workload	**Query Q12**			
Pattern	**Oracle** (26.85sec)		**ICE**(6sec)	
	Rows	I/O Costs	Rows	I/O Costs
Data Access	1511.7K	33479.167	7515.9K	7475
Non-vector	1511.7K	33479.167	7515.9K	7475
Tuple Reconstruction			30.9K	65
Group By	11.7K	726.281	30.9K	65
Projection	2	0.126	2	65

Table A.10.: *Accessed data of TPC-H query Q12 - Number of rows and I/O costs in KBytes.*

Workload	**Query Q13**			
Pattern	**Oracle** (5.24sec)		**ICE** (22sec)	
	Rows	I/O Costs	Rows	I/O Costs
Data Access	1575K	87625	1699.3K	1690
Non-vector	1575K	87625	1699.3K	1690
Tuple Reconstruction			1483.9K	1495
Group By	201.4K	7956.248	1533.9K	195
Sort	100.7K	1309.256	150k	195
Projection	201.4K	2618.512	42	65

Table A.11.: *Accessed data of TPC-H query Q13 - Number of rows and I/O costs in KBytes.*

Workload	**Query Q14**			
Pattern	**Oracle** (22.55sec)		**ICE** (3sec)	
	Rows	I/O Cost	Rows	I/O Cost
Data Access	273.7K	7020.894	6274.1K	6240
Non-vector	273.7K	7020.894	6274.1K	6240
Tuple Reconstruction			75.9K	5980
Sum	73.7K	3610.173	75.9K	5980
Projection	1	0.049	1	65

Table A.12.: *Accessed data of TPC-H query Q14 - Number of rows and I/O cost in KBytes.*

Workload	**Query Q18**			
Pattern	**Oracle** (31.56sec)		**ICE** (9sec)	
	Rows	I/O Costs	Rows	I/O Costs
Data Access	6001.2K	54011.02	7712K	7670
Non-vector	32	0.425	9215.2K	9165
Tuple Reconstruction			798	65
Group By	6001.2K	54010.935	399	65
Sort	5	0.265	57	65
Count	4	0.3	57	65
Projection	9	0.562	57	65

Table A.13.: *Accessed data of TPC-H query Q18 - Number of rows and I/O costs in KBytes.*

Workload	**Query Q19**			
Pattern	**Oracle** (33.27sec)		**ICE** (11sec)	
	Rows	I/O Costs	Rows	I/O Costs
Data Access	239.1K	12899.256	18822.5K	18720
Non-vector	239.1K	12899.256	6274.2K	6240
Tuple Reconstruction			121	65
Filtering			96	130
Sum	357	29.988	121	65
Projection	1	0.084	1	65

Table A.14.: *Accessed data of TPC-H query Q19 - Number of rows and I/O costs in KBytes.*

Workload	**Query Q20**			
Pattern	**Oracle** (31.56sec)		**ICE** (9min 43sec)	
	Rows	I/O Costs	Rows	I/O Costs
Data Access	869.9K	17397.8	7123.8K	7085
Non-vector	869.9K	17398.274	130.7K	130
Group By	4	0.388		
Sort	1	0.092	204	65
Projection	2	0.096	204	65

Table A.15.: *Accessed data of TPC-H query Q20 - Number of rows and I/O costs in KBytes.*

Workload	**Query Q21**			
Pattern	**Oracle** (1min 2.48sec)		**ICE** (6h 3min 58sec)	
	Rows	I/O Costs	Rows	I/O Costs
Data Access	6511.2K	166481.649	13659.4K	13585
Non-vector	13218.7K	547594.066	15162.6K	15080
Tuple Reconstruction			12.9K	65
Group By	177.1K	32685.07	4141	65
Sort	3043K	115632.914	411	65
Count	100	4	411	65
Projection	300	23.6	100	65

Table A.16.: *Accessed data of TPC-H query Q21 - Number of rows and I/O costs in KBytes.*

Workload	**Query Q22**			
Pattern	**Oracle** (5.34sec)		**ICE** (1sec)	
	Rows	I/O Costs	Rows	I/O Costs
Data Access	1509.8K	7717.578	392.1K	390
Non-vector	1500.5K	7513.77		
Filtering			588.2K	585
Group By	5	0.160	6384	65
Count	9.3K	203.808	7	65
Projection	1	0.032	7	65

Table A.17.: *Accessed data of TPC-H query Q22 - Number of rows and I/O costs in KBytes.*

A.3. TPC-H Queries and TPC-C Transaction

In this section, we present TPC-H queries [Tra10] and TPC-C transactions [Fer06], which we used for our evaluations and not presented in Chapters 4, 5, or 6. Queries for the TPC-CH benchmark can be found online: `https://db.in.tum.de/research/projects/CHbenCHmark/`.

```
1 SELECT l_returnflag,l_linestatus,SUM(l_quantity) AS sum_qty,SUM(l_extendedprice) AS
    sum_base_price,SUM(l_extendedprice * (1 - l_discount)) AS sum_disc_price,SUM(l_extendedprice *
    (1 - l_discount) * (1 + l_tax)) AS sum_charge,AVG(l_quantity) AS avg_qty,AVG(l_extendedprice) AS
    avg_price,AVG(l_discount) AS avg_disc,COUNT(*) AS count_order
2 FROM lineitem
3 WHERE l_shipdate <= date '1998-12-01' - interval '90' day (3)
4 GROUP BY l_returnflag,l_linestatus ORDER BY l_returnflag,l_linestatus;
```

Figure A.15.: *TPC-H query Q1 [Tra10].*

```
1 SELECT * FROM (
2     SELECT s_acctbal,s_name,n_name,p_partkey,p_mfgr,s_address,s_phone,s_comment
3     FROM part,supplier,partsupp,nation,region
4     WHERE p_partkey = ps_partkey AND s_suppkey = ps_suppkey AND p_size = 15 AND p_type
        LIKE '%BRASS' AND s_nationkey = n_nationkey AND n_regionkey = r_regionkey AND r_name =
        'EUROPE' AND ps_supplycost = (
5         SELECT MIN(ps_supplycost)
6         FROM partsupp,supplier,nation,region
7         WHERE p_partkey = ps_partkey AND s_suppkey = ps_suppkey AND s_nationkey =
            n_nationkey AND n_regionkey = r_regionkey AND r_name = 'EUROPE'
8     ) ORDER BY s_acctbal desc,n_name,s_name,p_partkey)
9 WHERE rownum <= 100;
```

Figure A.16.: *TPC-H query Q2 [Tra10].*

```
1 SELECT * FROM (
2     SELECT l_orderkey,SUM(l_extendedprice * (1 - l_discount)) AS revenue,o_orderdate,o_shippriority
3     FROM customer,orders,lineitem
4     WHERE c_mktsegment = 'BUILDING' AND c_custkey = o_custkey AND l_orderkey = o_orderkey
        AND o_orderdate < date '1995-03-15' AND l_shipdate > date '1995-03-15'
5 GROUP BY l_orderkey,o_orderdate,o_shippriority ORDER BY revenue DESC,o_orderdate)
6 WHERE rownum <= 10;
```

Figure A.17.: *TPC-H query Q3 [Tra10].*

```
1 SELECT n_name,sum(l_extendedprice * (1 - l_discount)) as revenue
2 FROM customer,orders,lineitem,supplier,nation,region
3 WHERE c_custkey = o_custkey AND l_orderkey = o_orderkey AND l_suppkey = s_suppkey AND
    c_nationkey = s_nationkey AND s_nationkey = n_nationkey AND n_regionkey = r_regionkey AND
    r_name = 'ASIA' AND o_orderdate >= date '1994-01-01' AND o_orderdate < date '1994-01-01' +
    interval '1' year
4 GROUP BY n_name ORDER BY revenue DESC;
```

Figure A.18.: *TPC-H query Q5 [Tra10].*

```
1 SELECT supp_nation,cust_nation,l_year,SUM(volume) AS revenue
2 FROM (SELECT n1.n_name AS supp_nation,n2.n_name AS cust_nation,EXTRACT(year FROM
    l_shipdate) AS l_year,l_extendedprice * (1 - l_discount) AS volume
3     FROM supplier,lineitem,orders,customer,nation n1,nation n2
4     WHERE s_suppkey = l_suppkey AND o_orderkey = l_orderkey AND c_custkey = o_custkey AND
        s_nationkey = n1.n_nationkey AND c_nationkey = n2.n_nationkey AND (
5         (n1.n_name = 'FRANCE' AND n2.n_name = 'GERMANY') OR (n1.n_name = 'GERMANY' AND
            n2.n_name = 'FRANCE'))
6     AND l_shipdate between date '1995-01-01' AND date '1996-12-31') shipping
7 GROUP BY supp_nation,cust_nation,l_year ORDER BY supp_nation,cust_nation,l_year;
```

Figure A.19.: *TPC-H query Q7 [Tra10].*

```
1 SELECT o_year,SUM(CASE WHEN nation = 'BRAZIL' THEN volume ELSE 0 END) / SUM(volume) AS
    mkt_share
2 FROM (SELECT EXTRACT(year FROM o_orderdate) AS o_year,l_extendedprice * (1 - l_discount) AS
    volume,n2.n_name AS nation
3     FROM part,supplier,lineitem,orders,customer,nation n1,nation n2,region
4     WHERE p_partkey = l_partkey AND s_suppkey = l_suppkey AND l_orderkey = o_orderkey AND
        o_custkey = c_custkey AND c_nationkey = n1.n_nationkey AND n1.n_regionkey = r_regionkey
        AND r_name = 'AMERICA' AND s_nationkey = n2.n_nationkey AND o_orderdate BETWEEN
        date '1995-01-01' AND date '1996-12-31' AND p_type = 'ECONOMY ANODIZED STEEL')
        all_nations
5 GROUP BY o_year ORDER BY o_year;
```

Figure A.20.: *TPC-H query Q8 [Tra10].*

```
1 SELECT nation,o_year,SUM(amount) AS sum_profit
2 FROM (SELECT n_name nation,EXTRACT(year FROM o_orderdate) o_year,l_extendedprice * (1 -
    l_discount) - ps_supplycost * l_quantity amount
3 FROM part,supplier,lineitem,partsupp,orders,nation
4 WHERE s_suppkey = l_suppkey AND ps_suppkey = l_suppkey AND ps_partkey = l_partkey AND
    p_partkey = l_partkey AND o_orderkey = l_orderkey AND s_nationkey = n_nationkey AND p_name
    LIKE '%green%') profit
5 GROUP BY nation,o_year ORDER BY nation,o_year DESC;
```

Figure A.21.: *TPC-H query Q9 [Tra10].*

```
1 SELECT * FROM (
2     SELECT c_custkey,c_name,SUM(l_extendedprice * (1 - l_discount)) AS
        revenue,c_acctbal,n_name,c_address,c_phone,c_comment
3     FROM customer,orders,lineitem,nation
4     WHERE c_custkey = o_custkey and l_orderkey = o_orderkey and o_orderdate >= date
        '1993-10-01' and o_orderdate < date '1993-10-01' + interval '3' month and l_returnflag = 'R'
        and c_nationkey = n_nationkey
5 GROUP BY c_custkey,c_name,c_acctbal,c_phone,n_name,c_address,c_comment ORDER BY revenue
    DESC)
6 WHERE rownum <= 20;
```

Figure A.22.: *TPC-H query Q10 [Tra10].*

```
1 SELECT ps_partkey,SUM(ps_supplycost * ps_availqty) AS value
2 FROM partsupp,supplier,nation
3 WHERE ps_suppkey = s_suppkey AND s_nationkey = n_nationkey AND n_name = 'GERMANY'
4 GROUP BY ps_partkey HAVING SUM(ps_supplycost * ps_availqty) > (
5     SELECT SUM(ps_supplycost * ps_availqty) * 0.0001000000
6     FROM partsupp,supplier,nation
7     WHERE ps_suppkey = s_suppkey AND s_nationkey = n_nationkey AND n_name = 'GERMANY')
8 ORDER BY value DESC;
```

Figure A.23.: *TPC-H query Q11 [Tra10].*

```
1 SELECT l_shipmode,SUM(CASE WHEN o_orderpriority = '1-URGENT' OR o_orderpriority = '2-HIGH'
    THEN 1 ELSE 0 END) as high_line_count,SUM(CASE WHEN o_orderpriority <> '1-URGENT' AND
    o_orderpriority <> '2-HIGH' THEN 1 ELSE 0 END) as low_line_count
2 FROM orders,lineitem
3 WHERE o_orderkey = l_orderkey AND l_shipmode in ('MAIL', 'SHIP') AND l_commitdate <
    l_receiptdate AND l_shipdate < l_commitdate AND l_receiptdate >= date '1994-01-01' AND
    l_receiptdate < date '1994-01-01' + interval '1' year
4 GROUP BY l_shipmode ORDER BY l_shipmode;
```

Figure A.24.: *TPC-H query Q12 [Tra10].*

```
1 SELECT 100.00 * SUM(CASE WHEN p_type LIKE 'PROMO%' THEN l_extendedprice * (1 -
    l_discount) ELSE 0 END) / SUM(l_extendedprice * (1 - l_discount)) AS promo_revenue
2 FROM lineitem,part
3 WHERE l_partkey = p_partkey AND l_shipdate >= date '1995-09-01' AND l_shipdate < date
    '1995-09-01' + interval '1' month;
```

Figure A.25.: *TPC-H query Q14 [Tra10].*

```
1  CREATE VIEW revenue0 (supplier_no, total_revenue) AS
2      SELECT l_suppkey,SUM(l_extendedprice * (1 - l_discount))
3      FROM lineitem
4      WHERE l_shipdate >= date '1996-01-01' AND l_shipdate < date '1996-01-01' + interval '3'
         month
5      GROUP BY l_suppkey;
6
7  SELECT s_suppkey,s_name,s_address,s_phone,total_revenue
8  FROM supplier,revenue0
9  WHERE s_suppkey = supplier_no AND total_revenue = (
10     SELECT MAX(total_revenue) FROM revenue0)
11 ORDER BY s_suppkey;
12
13 DROP VIEW revenue0;
```

Figure A.26.: *TPC-H query Q15 [Tra10].*

```
1 SELECT p_brand,p_type,p_size,COUNT(DISTINCT ps_suppkey) AS supplier_cnt
2 FROM partsupp,part
3 WHERE p_partkey = ps_partkey AND p_brand <> 'Brand#45' AND p_type not LIKE 'MEDIUM
    POLISHED%' AND p_size in (49, 14, 23, 45, 19, 3, 36, 9) AND ps_suppkey NOT IN (
4     SELECT s_suppkey FROM supplier WHERE s_comment LIKE '%Customer%Complaints%')
5 GROUP BY p_brand,p_type,p_size ORDER BY supplier_cnt DESC, p_brand, p_type, p_size;
```

Figure A.27.: *TPC-H query Q16 [Tra10].*

```
1 SELECT * FROM (
2     SELECT c_name,c_custkey,o_orderkey,o_orderdate,o_totalprice,SUM(l_quantity)
3     FROM customer,orders,lineitem
4     WHERE o_orderkey IN (
5         SELECT l_orderkey FROM lineitem GROUP BY l_orderkey HAVING SUM(l_quantity) > 300)
6     AND c_custkey = o_custkey AND o_orderkey = l_orderkey
7 GROUP BY c_name,c_custkey,o_orderkey,o_orderdate,o_totalprice ORDER BY o_totalprice DESC,
    o_orderdate)
8 WHERE rownum <= 100;
```

Figure A.28.: *TPC-H query Q18 [Tra10].*

```
1 SELECT SUM(l_extendedprice* (1 - l_discount)) AS revenue
2 WHERE (
3     p_partkey = l_partkey AND p_brand = 'Brand#12' AND p_container in ('SM CASE', 'SM BOX', 'SM
        PACK', 'SM PKG') AND l_quantity >= 1 and l_quantity <= 1 + 10 AND p_size between 1 and 5
        AND l_shipmode in ('AIR', 'AIR REG') AND l_shipinstruct = 'DELIVER IN PERSON')
4 OR (
5     p_partkey = l_partkey AND p_brand = 'Brand#23' AND p_container in ('MED BAG', 'MED BOX',
        'MED PKG', 'MED PACK') AND l_quantity >= 10 and l_quantity <= 10 + 10 AND p_size
        between 1 and 10 AND l_shipmode in ('AIR', 'AIR REG') AND l_shipinstruct = 'DELIVER IN
        PERSON')
6 OR (
7     p_partkey = l_partkey AND p_brand = 'Brand#34' AND p_container in ('LG CASE', 'LG BOX', 'LG
        PACK', 'LG PKG') AND l_quantity >= 20 and l_quantity <= 20 + 10 AND p_size between 1 and
        15 AND l_shipmode in ('AIR', 'AIR REG') AND l_shipinstruct = 'DELIVER IN PERSON');
```

Figure A.29.: *TPC-H query Q19 [Tra10].*

```
1  SELECT s_name,s_address
2  FROM supplier,nation
3  WHERE s_suppkey IN (
4      SELECT ps_suppkey FROM partsupp WHERE ps_partkey IN (
5          SELECT p_partkey FROM part WHERE p_name LIKE 'forest%')
6      AND ps_availqty > (SELECT 0.5 * SUM(l_quantity) FROM lineitem
7          WHERE l_partkey = ps_partkey AND l_suppkey = ps_suppkey AND l_shipdate >= date
             '1994-01-01' AND l_shipdate < date '1994-01-01' + interval '1' year))
8  AND s_nationkey = n_nationkey AND n_name = 'CANADA'
9  ORDER BY s_name;
```

Figure A.30.: *TPC-H query Q20 [Tra10].*

```
1  SELECT * FROM (
2      SELECT s_name,COUNT(*) AS numwait
3      FROM supplier,lineitem l1,orders,nation
4      WHERE s_suppkey = l1.l_suppkey AND o_orderkey = l1.l_orderkey AND o_orderstatus = 'F' AND
         l1.l_receiptdate > l1.l_commitdate AND EXISTS (
5          SELECT * FROM lineitem l2 WHERE l2.l_orderkey = l1.l_orderkey AND l2.l_suppkey <>
             l1.l_suppkey)
6      AND NOT EXISTS (
7          SELECT * FROM lineitem l3 WHERE l3.l_orderkey = l1.l_orderkey AND l3.l_suppkey <>
             l1.l_suppkey AND l3.l_receiptdate > l3.l_commitdate)
8      AND s_nationkey = n_nationkey AND n_name = 'SAUDI ARABIA'
9  GROUP BY s_name ORDER BY numwait DESC,s_name)
10 WHERE rownum <= 100;
```

Figure A.31.: *TPC-H query Q21 [Tra10].*

```
1  SELECT cntrycode,COUNT(*) AS numcust,SUM(c_acctbal) AS totacctbal
2  FROM (SELECT SUBSTR(c_phone, 1, 2) AS cntrycode,c_acctbal
3          FROM customer
4          WHERE SUBSTR(c_phone, 1, 2) IN ('13', '31', '23', '29', '30', '18', '17') and c_acctbal > (
5              SELECT AVG(c_acctbal) FROM customer
6              WHERE c_acctbal > 0.00 AND SUBSTR(c_phone, 1, 2) IN ('13', '31', '23', '29', '30', '18',
                 '17'))
7          AND NOT EXISTS (SELECT * FROM orders WHERE o_custkey = c_custkey)
8      ) custsale
9  GROUP BY cntrycode ORDER BY cntrycode;
```

Figure A.32.: *TPC-H query Q22 [Tra10].*

```
1  SELECT d_next_o_id FROM district WHERE d_id = 1 AND d_w_id = 50;
2  SELECT COUNT(DISTINCT (s_i_id)) FROM stock, order_line WHERE ol_w_id = 50 AND ol_d_id =
     1 AND ol_o_id < 3001 AND ol_o_id >= 3001 -20 AND s_w_id = 50 AND s_i_id = ol_i_id AND
     s_quantity < 10;
3  COMMIT WORK;
```

Figure A.33.: *Single extracted transaction 2.8 (Delivery) [Fer06].*

```
1  SELECT w_tax, c_discount, c_last, c_credit FROM warehouse, customer WHERE w_id=17 AND
     c_w_id=17 AND c_d_id=7 AND c_id=1584;
2  SELECT d_next_o_id, d_tax FROM district WHERE d_id=7 AND d_w_id=17;
3  UPDATE district SET d_next_o_id=3001+1 WHERE d_id=7 AND d_w_id=17;
4  INSERT INTO orderr (o_id, o_d_id, o_w_id, o_c_id, o_entry_d, o_carrier_id, o_all_local) VALUES
     (3001,7,17,1584,TO_DATE('2012-05-19 18:06:00', 'YYYY/MM/DD HH24:MI:SS'),0,1);
5  INSERT INTO new_order (no_o_id, no_d_id, no_w_id) VALUES (3001,7,17);
6  SELECT i_price, i_name, i_data FROM item WHERE i_id=5576;
7  SELECT s_quantity, s_data, s_dist_01, s_dist_02, s_dist_03, s_dist_04, s_dist_05, s_dist_06,
     s_dist_07, s_dist_08, s_dist_09, s_dist_10 FROM stock WHERE s_i_id =5576 AND s_w_id =17;
8  UPDATE stock SET s_quantity=98 WHERE s_i_id =5576 AND s_w_id =17;
9  UPDATE stock SET s_ytd=0.000000 +3, s_order_cnt=0+1 WHERE s_i_id =5576 AND s_w_id =17;
10 INSERT INTO order_line (ol_o_id, ol_d_id, ol_w_id, ol_number, ol_i_id, ol_supply_w_id,
     ol_quantity, ol_amount, ol_dist_info) VALUES (3001, 7, 17, 1, 5576, 17, 3, 41.32, '6xB:fLK
     Hm;2=f2eWMwu7,]o');
11 SELECT i_price, i_name, i_data FROM item WHERE i_id=89016;
12 SELECT s_quantity, s_data, s_dist_01, s_dist_02, s_dist_03, s_dist_04, s_dist_05, s_dist_06,
     s_dist_07, s_dist_08, s_dist_09, s_dist_10 FROM stock WHERE s_i_id =89016 AND s_w_id =17;
13 UPDATE stock SET s_quantity=85 WHERE s_i_id =89016 AND s_w_id =17;
14 UPDATE stock SET s_ytd=0.000000 +1, s_order_cnt=0+1 WHERE s_i_id =89016 AND s_w_id =17;
15 INSERT INTO order_line (ol_o_id, ol_d_id, ol_w_id, ol_number, ol_i_id, ol_supply_w_id,
     ol_quantity, ol_amount, ol_dist_info) VALUES (3001, 7, 17, 2, 89016, 17, 1, 40.90,
     'HDcBT#T|G;B{>v{f5@dT:c=P');
16 SELECT i_price, i_name, i_data FROM item WHERE i_id=52205;
17 SELECT s_quantity, s_data, s_dist_01, s_dist_02, s_dist_03, s_dist_04, s_dist_05, s_dist_06,
     s_dist_07, s_dist_08, s_dist_09, s_dist_10 FROM stock WHERE s_i_id =52205 AND s_w_id =17;
18 UPDATE stock SET s_quantity=87 WHERE s_i_id =52205 AND s_w_id =17;
19 UPDATE stock SET s_ytd=0.000000 +7, s_order_cnt=0+1 WHERE s_i_id =52205 AND s_w_id =17;
20 INSERT INTO order_line (ol_o_id, ol_d_id, ol_w_id, ol_number, ol_i_id, ol_supply_w_id,
     ol_quantity, ol_amount, ol_dist_info) VALUES (3001, 7, 17, 3, 52205, 17, 7, 488.66,
     '\YqBa|aAYUc=UH_1!EOVn1ho');
21 SELECT i_price, i_name, i_data FROM item WHERE i_id=97160;
22 SELECT s_quantity, s_data, s_dist_01, s_dist_02, s_dist_03, s_dist_04, s_dist_05, s_dist_06,
     s_dist_07, s_dist_08, s_dist_09, s_dist_10 FROM stock WHERE s_i_id =97160 AND s_w_id =17;
23 UPDATE stock SET s_quantity=71 WHERE s_i_id =97160 AND s_w_id =17;
24 UPDATE stock SET s_ytd=0.000000 +3, s_order_cnt=0+1 WHERE s_i_id =97160 AND s_w_id =17;
25 INSERT INTO order_line (ol_o_id, ol_d_id, ol_w_id, ol_number, ol_i_id, ol_supply_w_id,
     ol_quantity, ol_amount, ol_dist_info) VALUES (3001, 7, 17, 4, 97160, 17, 3, 293.24,
     'k=."32t"P%$c8yeh"jN:f7%h');
26 SELECT i_price, i_name, i_data FROM item WHERE i_id=64374;
27 SELECT s_quantity, s_data, s_dist_01, s_dist_02, s_dist_03, s_dist_04, s_dist_05, s_dist_06,
     s_dist_07, s_dist_08, s_dist_09, s_dist_10 FROM stock WHERE s_i_id =64374 AND s_w_id =17;
28 UPDATE stock SET s_quantity=57 WHERE s_i_id =64374 AND s_w_id =17;
29 UPDATE stock SET s_ytd=0.000000 +5, s_order_cnt=0+1 WHERE s_i_id =64374 AND s_w_id =17;
30 INSERT INTO order_line (ol_o_id, ol_d_id, ol_w_id, ol_number, ol_i_id, ol_supply_w_id,
     ol_quantity, ol_amount, ol_dist_info) VALUES (3001, 7, 17, 5, 64374, 17, 5, 465.01,
     'kk-x+oFv]TrhF<b(qXp;tVCi');
31 UPDATE orderr SET o_ol_cnt=5 WHERE o_id=3001 AND o_d_id=7 AND o_w_id=17;
32 COMMIT WORK;
```

Figure A.34.: *Single extracted transaction 2.4 (NewOrder) [Fer06].*

```
1  SELECT w_name, w_street_1, w_street_2, w_city, w_state, w_zip FROM warehouse WHERE w_id =
     11;
2  UPDATE warehouse SET w_ytd = w_ytd + 1708.16 WHERE w_id = 11;
3  SELECT d_name, d_street_1, d_street_2, d_city, d_state, d_zip FROM district WHERE d_w_id = 11
     AND d_id = 8;
4  UPDATE district SET d_ytd = d_ytd + 1708.16 WHERE d_id = 8 AND d_w_id = 11;
5  SELECT count(c_id) FROM customer WHERE c_last = 'CALLIOUGTHATION' AND c_d_id = 9 AND
     c_w_id = 84;
6  DECLARE c_id customer.c_id%TYPE; c_first customer.c_first%TYPE; c_middle
     customer.c_middle%TYPE; c_street_1 customer.c_street_1%TYPE; c_street_2
     customer.c_street_2%TYPE; c_city customer.c_city%TYPE; c_state customer.c_state%TYPE; c_zip
     customer.c_zip%TYPE; c_phone customer.c_phone%TYPE; c_credit customer.c_credit%TYPE;
     c_credit_lim customer.c_credit_lim%TYPE; c_discount customer.c_discount%TYPE; c_balance
     customer.c_balance%TYPE; c_since customer.c_since%TYPE; CURSOR c_porlast IS SELECT c_id,
     c_first, c_middle, c_street_1, c_street_2, c_city, c_state, c_zip, c_phone, c_credit, c_credit_lim,
     c_discount, c_balance, c_since FROM customer WHERE c_w_id = 84 AND c_d_id = 9 AND c_last
     = 'CALLIOUGTHATION' ORDER BY c_first;
7  BEGIN OPEN c_porlast;
8  FETCH c_porlast INTO c_id, c_first, c_middle, c_street_1, c_street_2, c_city, c_state, c_zip, c_phone,
     c_credit, c_credit_lim, c_discount, c_balance, c_since;
9  CLOSE c_porlast; END;
10 UPDATE customer SET c_balance = c_balance − 1708.16, c_ytd_payment = c_ytd_payment + 1708.16,
     c_payment_cnt = c_payment_cnt +1 WHERE c_w_id = 84 AND c_d_id = 9 AND c_id = 719;
11 INSERT INTO history (h_c_d_id, h_c_w_id, h_c_id, h_d_id, h_w_id, h_date, h_amount, h_data)
     VALUES (9, 84, 719, 8, 11, TO_DATE('2012−05−19 18:05:49', 'YYYY/MM/DD HH24:MI:SS'), 1708.16,
     '8kA5DHz R|.Z^q:4q');
12 COMMIT WORK;
13 SELECT w_name, w_street_1, w_street_2, w_city, w_state, w_zip FROM warehouse WHERE w_id =
     10;
14 UPDATE warehouse SET w_ytd = w_ytd + 4701.98 WHERE w_id = 10;
15 SELECT d_name, d_street_1, d_street_2, d_city, d_state, d_zip FROM district WHERE d_w_id = 10
     AND d_id = 9;
16 UPDATE district SET d_ytd = d_ytd + 4701.98 WHERE d_id = 9 AND d_w_id = 10;
17 SELECT c_first, c_middle, c_last, c_street_1, c_street_2, c_city, c_state, c_zip, c_phone, c_credit,
     c_discount, c_balance, c_since FROM customer WHERE c_w_id = 10 AND c_d_id = 9 AND c_id =
     2781;
18 UPDATE customer SET c_balance = c_balance − 4701.98, c_ytd_payment = c_ytd_payment + 4701.98,
     c_payment_cnt = c_payment_cnt +1 WHERE c_w_id = 10 AND c_d_id = 9 AND c_id = 2781;
19 INSERT INTO history (h_c_d_id, h_c_w_id, h_c_id, h_d_id, h_w_id, h_date, h_amount, h_data)
     VALUES (9, 10, 2781, 9, 10, TO_DATE('2012−05−19 18:05:49', 'YYYY/MM/DD HH24:MI:SS'),
     4701.98, '^^A]\Vp= |4f(Ce%');
20 COMMIT WORK;
```

Figure A.35.: *Single extracted transaction 2.5 (Payment) [Fer06].*

```
1  SELECT c_balance, c_first, c_middle, c_last FROM customer WHERE c_w_id = 15 AND c_d_id = 9
     AND c_id = 1271;
2  DECLARE cur_ordenes CURSOR FOR SELECT o_id, o_entry_d, o_carrier_id FROM orderr WHERE
     o_w_id = 15 AND o_d_id = 9 AND o_c_id = 1271 ORDER BY o_id DESC;
3  OPEN cur_ordenes;
4  FETCH FROM cur_ordenes INTO o_id = 2296, o_entry_d = 2012-03-11 01:01:27, o_carrier_id = 0;
5  CLOSE cur_ordenes;
6  DECLARE cur_ord_lines CURSOR FOR SELECT ol_i_id, ol_supply_w_id, ol_quantity, ol_amount,
     ol_delivery_d FROM order_line WHERE ol_w_id = 15 AND ol_d_id = 9 AND ol_o_id = 2296;
7  AGDB~Execution OrderState.10: OPEN cur_ord_lines;
8  FETCH FROM cur_ord_lines INTO ol_i_id = 74255, ol_supply_w_id = 15, ol_quantity = 5, ol_amount
     = 4653.77, ol_delivery_d = 1970-01-01 00:00:00;
9  FETCH FROM cur_ord_lines INTO ol_i_id = 14379, ol_supply_w_id = 15, ol_quantity = 5, ol_amount
     = 640.04, ol_delivery_d = 1970-01-01 00:00:00;
10 FETCH FROM cur_ord_lines INTO ol_i_id = 22852, ol_supply_w_id = 15, ol_quantity = 5, ol_amount
     = 5825.25, ol_delivery_d = 1970-01-01 00:00:00;
11 FETCH FROM cur_ord_lines INTO ol_i_id = 59773, ol_supply_w_id = 15, ol_quantity = 5, ol_amount
     = 8991.15, ol_delivery_d = 1970-01-01 00:00:00;
12 FETCH FROM cur_ord_lines INTO ol_i_id = 41543, ol_supply_w_id = 15, ol_quantity = 5, ol_amount
     = 3106.39, ol_delivery_d = 1970-01-01 00:00:00;
13 FETCH FROM cur_ord_lines INTO ol_i_id = 35091, ol_supply_w_id = 15, ol_quantity = 5, ol_amount
     = 2181.42, ol_delivery_d = 1970-01-01 00:00:00;
14 FETCH FROM cur_ord_lines INTO ol_i_id = 16454, ol_supply_w_id = 15, ol_quantity = 5, ol_amount
     = 8192.52, ol_delivery_d = 1970-01-01 00:00:00;
15 FETCH FROM cur_ord_lines INTO ol_i_id = 15799, ol_supply_w_id = 15, ol_quantity = 5, ol_amount
     = 7760.15, ol_delivery_d = 1970-01-01 00:00:00;
16 FETCH FROM cur_ord_lines INTO ol_i_id = 49470, ol_supply_w_id = 15, ol_quantity = 5, ol_amount
     = 2821.46, ol_delivery_d = 1970-01-01 00:00:00;
17 FETCH FROM cur_ord_lines INTO ol_i_id = 39305, ol_supply_w_id = 15, ol_quantity = 5, ol_amount
     = 4017.79, ol_delivery_d = 1970-01-01 00:00:00;
18 FETCH FROM cur_ord_lines INTO ol_i_id = 75571, ol_supply_w_id = 15, ol_quantity = 5, ol_amount
     = 6751.31, ol_delivery_d = 1970-01-01 00:00:00;
19 FETCH FROM cur_ord_lines INTO ol_i_id = 91012, ol_supply_w_id = 15, ol_quantity = 5, ol_amount
     = 5927.84, ol_delivery_d = 1970-01-01 00:00:00;
20 FETCH FROM cur_ord_lines INTO ol_i_id = 74396, ol_supply_w_id = 15, ol_quantity = 5, ol_amount
     = 6199.20, ol_delivery_d = 1970-01-01 00:00:00;
21 FETCH FROM cur_ord_lines INTO ol_i_id = 92024, ol_supply_w_id = 15, ol_quantity = 5, ol_amount
     = 4943.85, ol_delivery_d = 1970-01-01 00:00:00;
22 FETCH FROM cur_ord_lines INTO ol_i_id = 92024, ol_supply_w_id = 15, ol_quantity = 5, ol_amount
     = 4943.85, ol_delivery_d = 1970-01-01 00:00:00;
23 CLOSE cur_ord_lines;
24 COMMIT WORK;
```

Figure A.36.: *Single extracted transaction 2.6 (StockLevel) [Fer06].*

```
1  SELECT min(no_o_id) FROM new_order WHERE no_w_id = 13 AND no_d_id = 1;
2  DELETE FROM new_order WHERE no_o_id = 2101 AND no_w_id = 13 AND no_d_id = 1;
3  SELECT o_c_id FROM orderr WHERE o_w_id = 13 AND o_d_id = 1 AND o_id = 2101;
4  UPDATE orderr SET o_carrier_id = 3 WHERE o_w_id = 13 AND o_d_id = 1 AND o_id = 2101;
5  UPDATE order_line SET ol_delivery_d = TO_DATE('2012-05-19 18:05:44', 'YYYY/MM/DD HH24:MI:SS') WHERE ol_o_id
     = 2101 AND ol_w_id = 13 AND ol_d_id = 1;
6  SELECT sum(ol_amount) FROM order_line WHERE ol_o_id = 2101 AND ol_w_id = 13 AND ol_d_id = 1;
7  UPDATE customer SET c_balance = c_balance + 27304.74, c_delivery_cnt = c_delivery_cnt + 1 WHERE c_w_id = 13 AND
     c_d_id = 1 AND c_id = 418;
8  SELECT min(no_o_id) FROM new_order WHERE no_w_id = 13 AND no_d_id = 2;
9  DELETE FROM new_order WHERE no_o_id = 2101 AND no_w_id = 13 AND no_d_id = 2;
10 SELECT o_c_id FROM orderr WHERE o_w_id = 13 AND o_d_id = 2 AND o_id = 2101;
11 UPDATE orderr SET o_carrier_id = 3 WHERE o_w_id = 13 AND o_d_id = 2 AND o_id = 2101;
12 UPDATE order_line SET ol_delivery_d = TO_DATE('2012-05-19 18:05:44', 'YYYY/MM/DD HH24:MI:SS') WHERE ol_o_id
     = 2101 AND ol_w_id = 13 AND ol_d_id = 2;
13 SELECT sum(ol_amount) FROM order_line WHERE ol_o_id = 2101 AND ol_w_id = 13 AND ol_d_id = 2;
14 UPDATE customer SET c_balance = c_balance + 64079.01, c_delivery_cnt = c_delivery_cnt + 1 WHERE c_w_id = 13 AND
     c_d_id = 2 AND c_id = 418;
15 SELECT min(no_o_id) FROM new_order WHERE no_w_id = 13 AND no_d_id = 3;
16 DELETE FROM new_order WHERE no_o_id = 2101 AND no_w_id = 13 AND no_d_id = 3;
17 SELECT o_c_id FROM orderr WHERE o_w_id = 13 AND o_d_id = 3 AND o_id = 2101;
18 UPDATE orderr SET o_carrier_id = 3 WHERE o_w_id = 13 AND o_d_id = 3 AND o_id = 2101;
19 UPDATE order_line SET ol_delivery_d = TO_DATE('2012-05-19 18:05:44', 'YYYY/MM/DD HH24:MI:SS') WHERE ol_o_id
     = 2101 AND ol_w_id = 13 AND ol_d_id = 3;
20 SELECT sum(ol_amount) FROM order_line WHERE ol_o_id = 2101 AND ol_w_id = 13 AND ol_d_id = 3;
21 UPDATE customer SET c_balance = c_balance + 27505.93, c_delivery_cnt = c_delivery_cnt + 1 WHERE c_w_id = 13 AND
     c_d_id = 3 AND c_id = 418;
22 SELECT min(no_o_id) FROM new_order WHERE no_w_id = 13 AND no_d_id = 4;
23 DELETE FROM new_order WHERE no_o_id = 2101 AND no_w_id = 13 AND no_d_id = 4;
24 SELECT o_c_id FROM orderr WHERE o_w_id = 13 AND o_d_id = 4 AND o_id = 2101;
25 UPDATE orderr SET o_carrier_id = 3 WHERE o_w_id = 13 AND o_d_id = 4 AND o_id = 2101;
26 UPDATE order_line SET ol_delivery_d = TO_DATE('2012-05-19 18:05:44', 'YYYY/MM/DD HH24:MI:SS') WHERE ol_o_id
     = 2101 AND ol_w_id = 13 AND ol_d_id = 4;
27 SELECT sum(ol_amount) FROM order_line WHERE ol_o_id = 2101 AND ol_w_id = 13 AND ol_d_id = 4;
28 UPDATE customer SET c_balance = c_balance + 46320.51, c_delivery_cnt = c_delivery_cnt + 1 WHERE c_w_id = 13 AND
     c_d_id = 4 AND c_id = 418;
29 SELECT min(no_o_id) FROM new_order WHERE no_w_id = 13 AND no_d_id = 5;
30 DELETE FROM new_order WHERE no_o_id = 2101 AND no_w_id = 13 AND no_d_id = 5;
31 SELECT o_c_id FROM orderr WHERE o_w_id = 13 AND o_d_id = 5 AND o_id = 2101;
32 UPDATE orderr SET o_carrier_id = 3 WHERE o_w_id = 13 AND o_d_id = 5 AND o_id = 2101;
33 UPDATE order_line SET ol_delivery_d = TO_DATE('2012-05-19 18:05:44', 'YYYY/MM/DD HH24:MI:SS') WHERE ol_o_id
     = 2101 AND ol_w_id = 13 AND ol_d_id = 5;
34 SELECT sum(ol_amount) FROM order_line WHERE ol_o_id = 2101 AND ol_w_id = 13 AND ol_d_id = 5;
35 UPDATE customer SET c_balance = c_balance + 67651.09, c_delivery_cnt = c_delivery_cnt + 1 WHERE c_w_id = 13 AND
     c_d_id = 5 AND c_id = 418;
36 SELECT min(no_o_id) FROM new_order WHERE no_w_id = 13 AND no_d_id = 6;
37 DELETE FROM new_order WHERE no_o_id = 2101 AND no_w_id = 13 AND no_d_id = 6;
38 SELECT o_c_id FROM orderr WHERE o_w_id = 13 AND o_d_id = 6 AND o_id = 2101;
39 UPDATE orderr SET o_carrier_id = 3 WHERE o_w_id = 13 AND o_d_id = 6 AND o_id = 2101;
40 UPDATE order_line SET ol_delivery_d = TO_DATE('2012-05-19 18:05:44', 'YYYY/MM/DD HH24:MI:SS') WHERE ol_o_id
     = 2101 AND ol_w_id = 13 AND ol_d_id = 6;
41 SELECT sum(ol_amount) FROM order_line WHERE ol_o_id = 2101 AND ol_w_id = 13 AND ol_d_id = 6;
42 UPDATE customer SET c_balance = c_balance + 66155.56, c_delivery_cnt = c_delivery_cnt + 1 WHERE c_w_id = 13 AND
     c_d_id = 6 AND c_id = 418;
43 SELECT min(no_o_id) FROM new_order WHERE no_w_id = 13 AND no_d_id = 7;
44 DELETE FROM new_order WHERE no_o_id = 2101 AND no_w_id = 13 AND no_d_id = 7;
45 SELECT o_c_id FROM orderr WHERE o_w_id = 13 AND o_d_id = 7 AND o_id = 2101;
46 UPDATE orderr SET o_carrier_id = 3 WHERE o_w_id = 13 AND o_d_id = 7 AND o_id = 2101;
47 UPDATE order_line SET ol_delivery_d = TO_DATE('2012-05-19 18:05:44', 'YYYY/MM/DD HH24:MI:SS') WHERE ol_o_id
     = 2101 AND ol_w_id = 13 AND ol_d_id = 7;
48 SELECT sum(ol_amount) FROM order_line WHERE ol_o_id = 2101 AND ol_w_id = 13 AND ol_d_id = 7;
49 UPDATE customer SET c_balance = c_balance + 32115.57, c_delivery_cnt = c_delivery_cnt + 1 WHERE c_w_id = 13 AND
     c_d_id = 7 AND c_id = 418;
50 SELECT min(no_o_id) FROM new_order WHERE no_w_id = 13 AND no_d_id = 8;
51 DELETE FROM new_order WHERE no_o_id = 2101 AND no_w_id = 13 AND no_d_id = 8;
52 SELECT o_c_id FROM orderr WHERE o_w_id = 13 AND o_d_id = 8 AND o_id = 2101;
53 UPDATE orderr SET o_carrier_id = 3 WHERE o_w_id = 13 AND o_d_id = 8 AND o_id = 2101;
54 UPDATE order_line SET ol_delivery_d = TO_DATE('2012-05-19 18:05:44', 'YYYY/MM/DD HH24:MI:SS') WHERE ol_o_id
     = 2101 AND ol_w_id = 13 AND ol_d_id = 8;
55 SELECT sum(ol_amount) FROM order_line WHERE ol_o_id = 2101 AND ol_w_id = 13 AND ol_d_id = 8;
56 UPDATE customer SET c_balance = c_balance + 63714.72, c_delivery_cnt = c_delivery_cnt + 1 WHERE c_w_id = 13 AND
     c_d_id = 8 AND c_id = 418;
57 SELECT min(no_o_id) FROM new_order WHERE no_w_id = 13 AND no_d_id = 9;
58 DELETE FROM new_order WHERE no_o_id = 2101 AND no_w_id = 13 AND no_d_id = 9;
59 SELECT o_c_id FROM orderr WHERE o_w_id = 13 AND o_d_id = 9 AND o_id = 2101;
60 UPDATE orderr SET o_carrier_id = 3 WHERE o_w_id = 13 AND o_d_id = 9 AND o_id = 2101;
61 UPDATE order_line SET ol_delivery_d = TO_DATE('2012-05-19 18:05:44', 'YYYY/MM/DD HH24:MI:SS') WHERE ol_o_id
     = 2101 AND ol_w_id = 13 AND ol_d_id = 9;
62 SELECT sum(ol_amount) FROM order_line WHERE ol_o_id = 2101 AND ol_w_id = 13 AND ol_d_id = 9;
63 UPDATE customer SET c_balance = c_balance + 34407.62, c_delivery_cnt = c_delivery_cnt + 1 WHERE c_w_id = 13 AND
     c_d_id = 9 AND c_id = 418;
64 SELECT min(no_o_id) FROM new_order WHERE no_w_id = 13 AND no_d_id = 10;
65 DELETE FROM new_order WHERE no_o_id = 2101 AND no_w_id = 13 AND no_d_id = 10;
66 SELECT o_c_id FROM orderr WHERE o_w_id = 13 AND o_d_id = 10 AND o_id = 2101;
67 UPDATE orderr SET o_carrier_id = 3 WHERE o_w_id = 13 AND o_d_id = 10 AND o_id = 2101;
68 UPDATE order_line SET ol_delivery_d = TO_DATE('2012-05-19 18:05:44', 'YYYY/MM/DD HH24:MI:SS') WHERE ol_o_id
     = 2101 AND ol_w_id = 13 AND ol_d_id = 10;
69 SELECT sum(ol_amount) FROM order_line WHERE ol_o_id = 2101 AND ol_w_id = 13 AND ol_d_id = 10;
70 UPDATE customer SET c_balance = c_balance + 36404.44, c_delivery_cnt = c_delivery_cnt + 1 WHERE c_w_id = 13 AND
     c_d_id = 10 AND c_id = 418; COMMIT WORK;
```


Figure A.37.: *Single extracted transaction 2.7 (OrderState) [Fer06].*

A.4. Resource Consumption of the Replicated Solution for TPC-H and TPC-C

We present the resource consumption for the remaining queries in the following, which we did not present in Chapter 6.

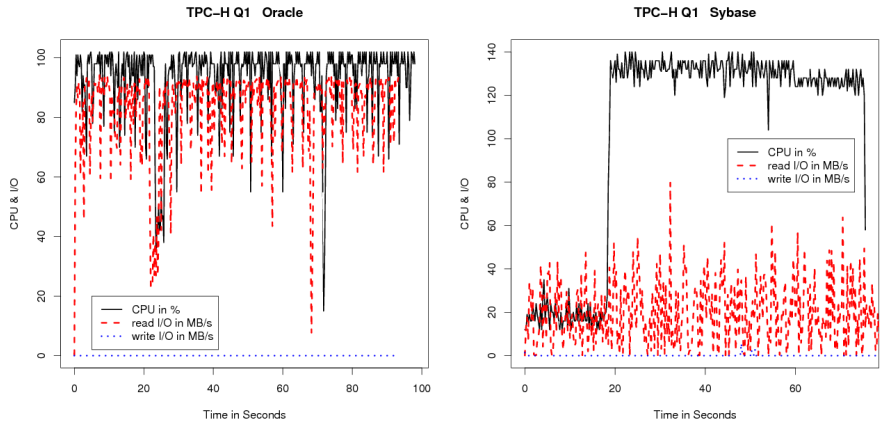


Figure A.38.: *CPU and I/O for TPC-H Q1 on Oracle and Sybase.*

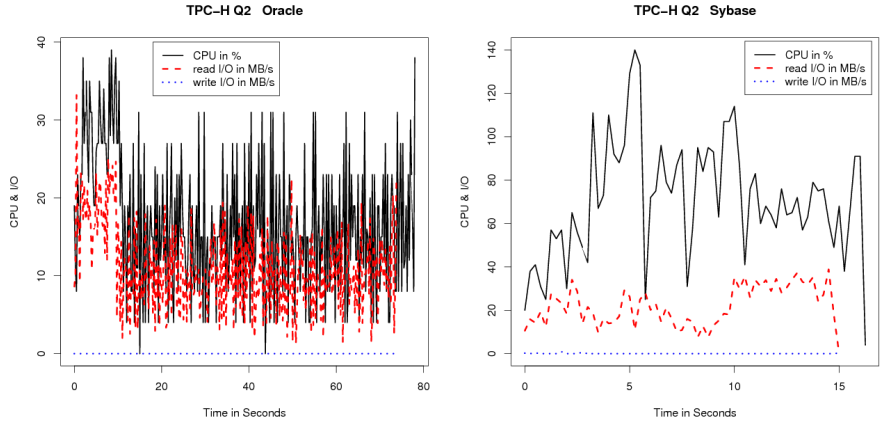


Figure A.39.: *CPU and I/O for TPC-H Q2 on Oracle and Sybase.*

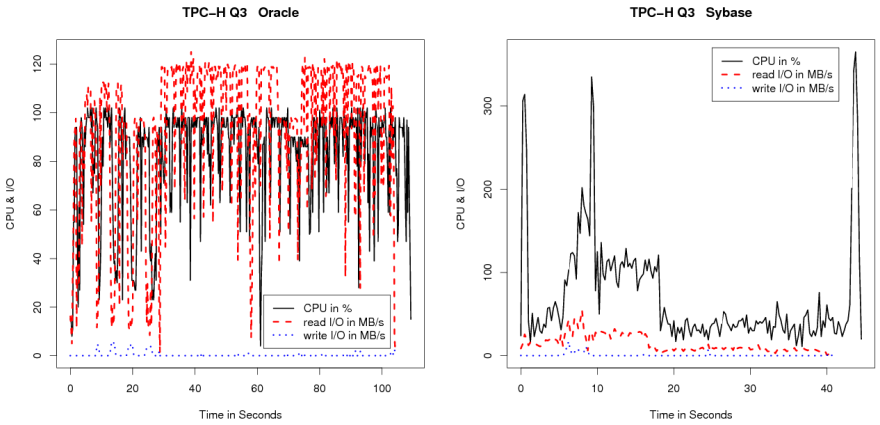


Figure A.40.: *CPU and I/O for TPC-H Q3 on Oracle and Sybase.*

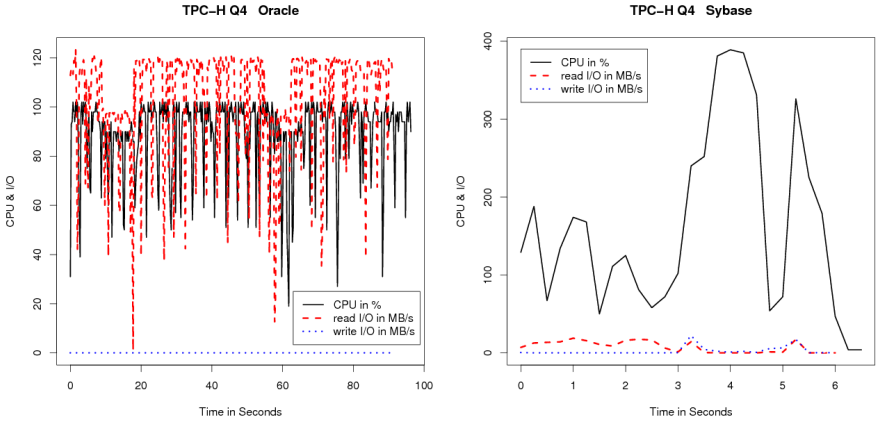


Figure A.41.: *CPU and I/O for TPC-H Q4 on Oracle and Sybase.*

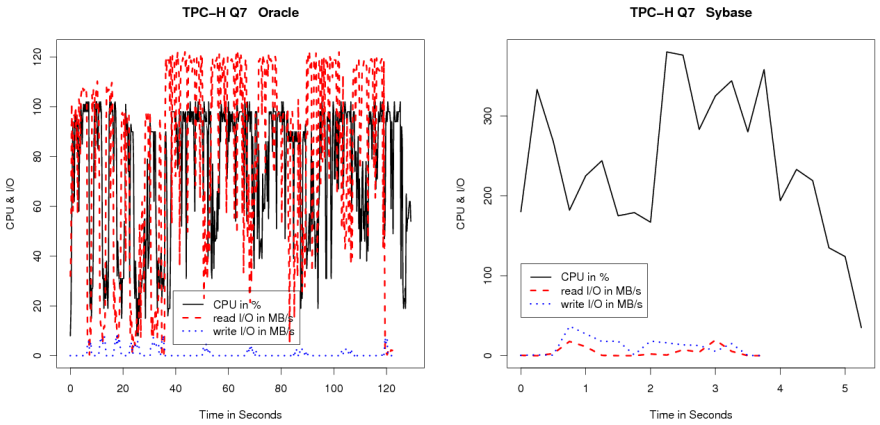


Figure A.42.: *CPU and I/O for TPC-H Q7 on Oracle and Sybase.*

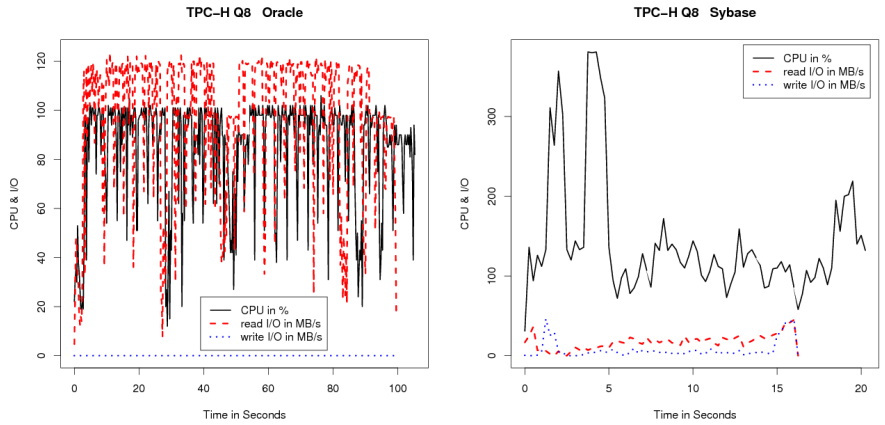


Figure A.43.: *CPU and I/O for TPC-H Q8 on Oracle and Sybase.*

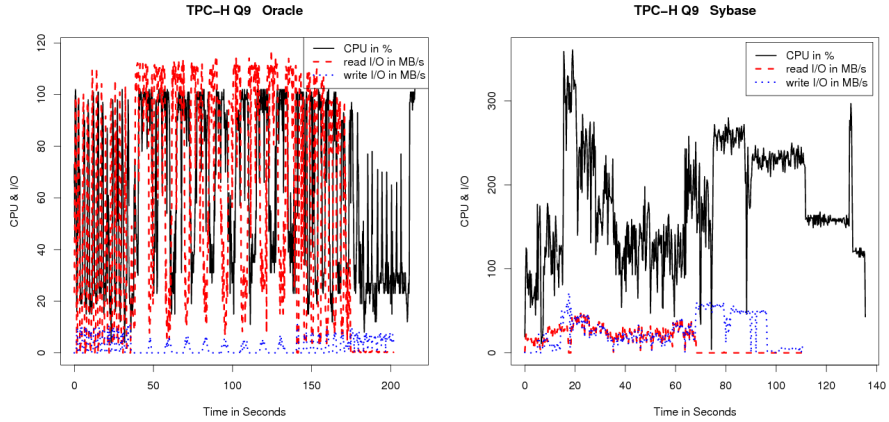


Figure A.44.: *CPU and I/O for TPC-H Q9 on Oracle and Sybase.*

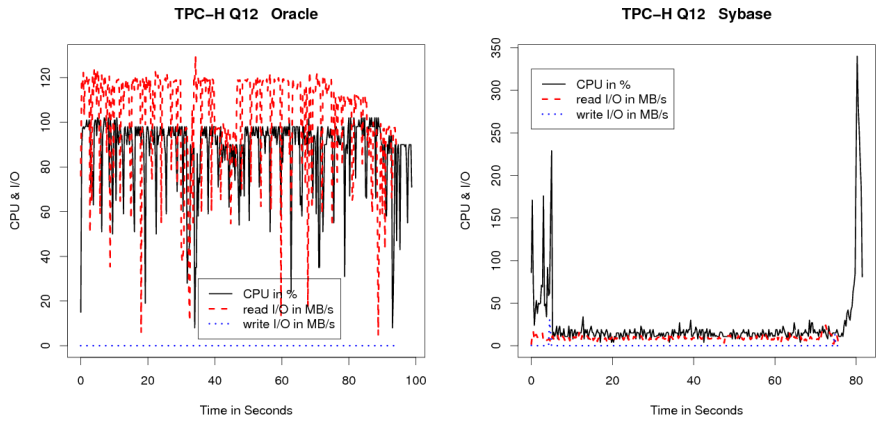


Figure A.45.: *CPU and I/O for TPC-H Q12 on Oracle and Sybase.*

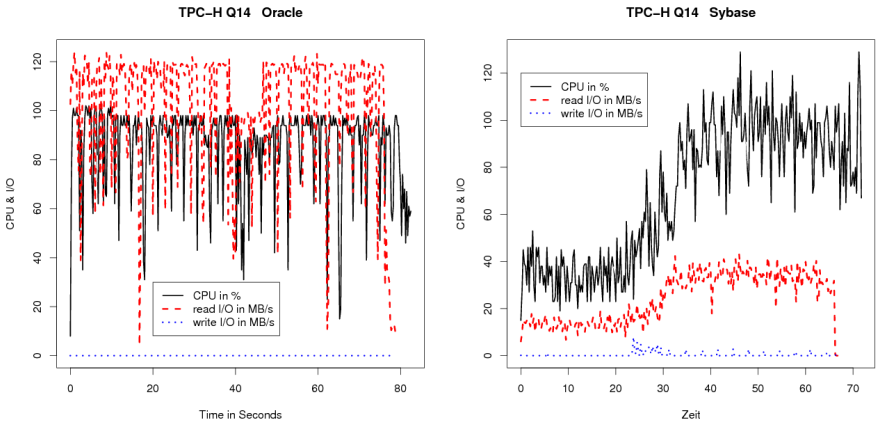


Figure A.46.: *CPU and I/O for TPC-H Q14 on Oracle and Sybase.*

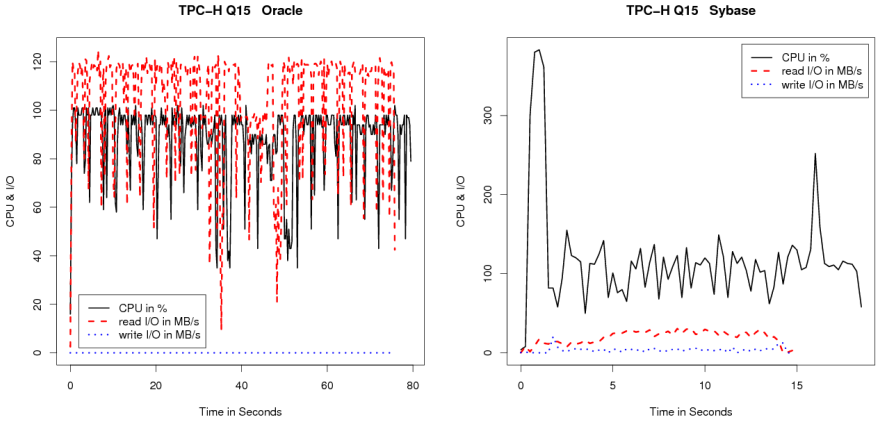


Figure A.47.: *CPU and I/O for TPC-H Q15 on Oracle and Sybase.*

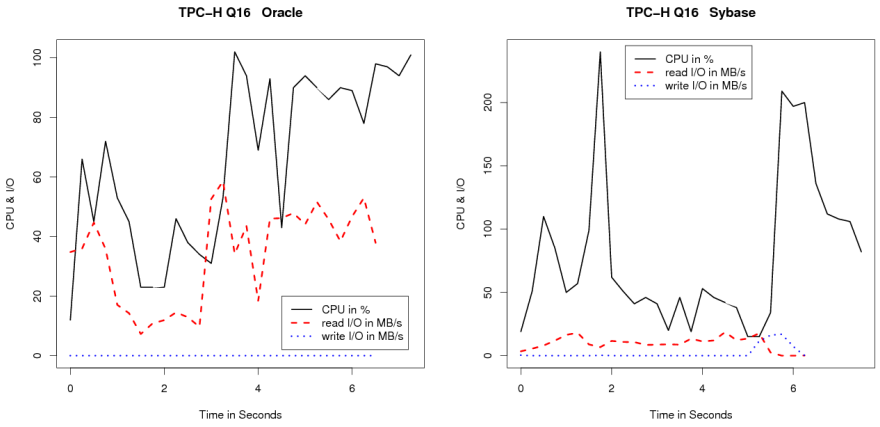


Figure A.48.: *CPU and I/O for TPC-H Q16 on Oracle and Sybase.*

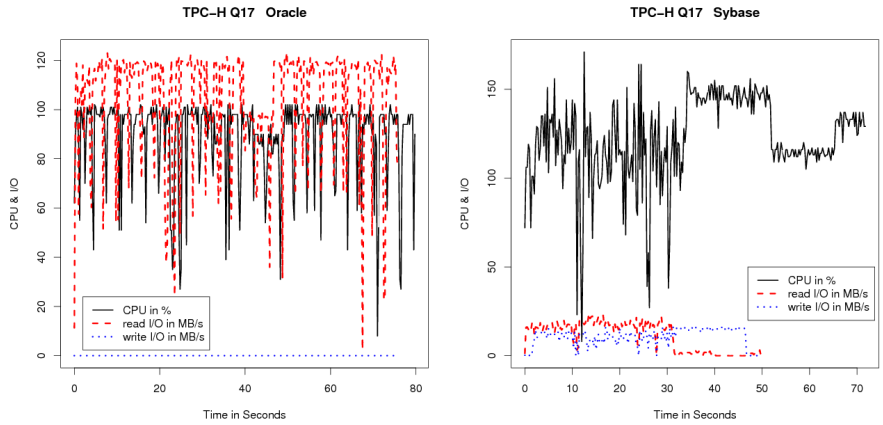


Figure A.49.: *CPU and I/O for TPC-H Q17 on Oracle and Sybase.*

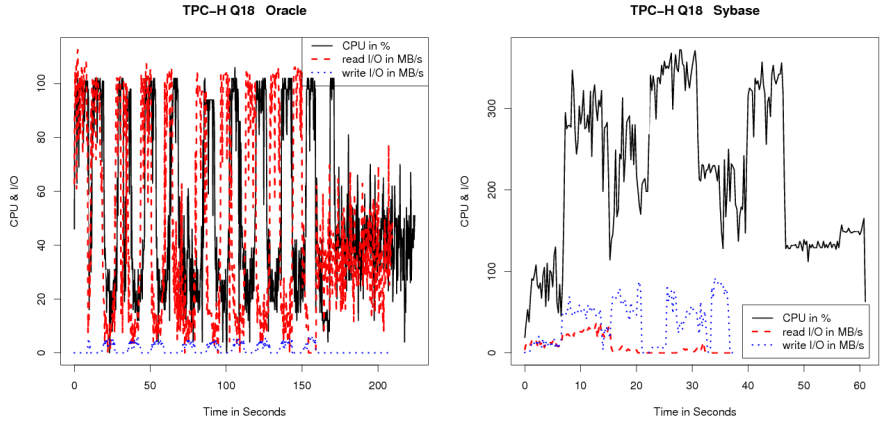


Figure A.50.: *CPU and I/O for TPC-H Q18 on Oracle and Sybase.*

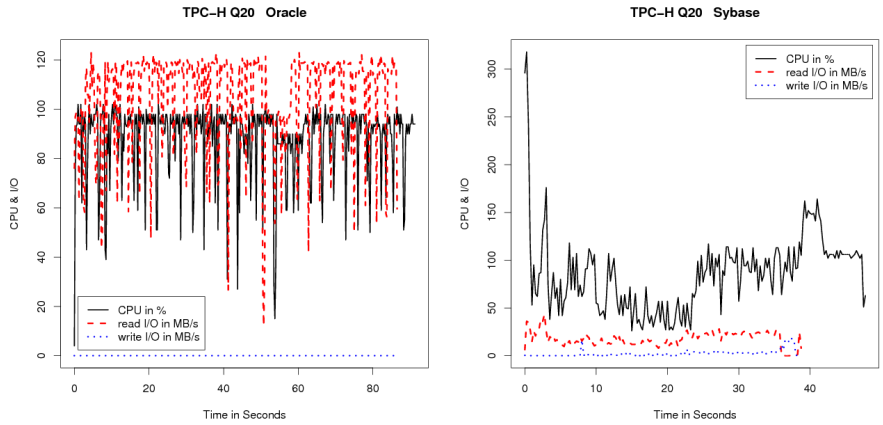


Figure A.51.: *CPU and I/O for TPC-H Q20 on Oracle and Sybase.*

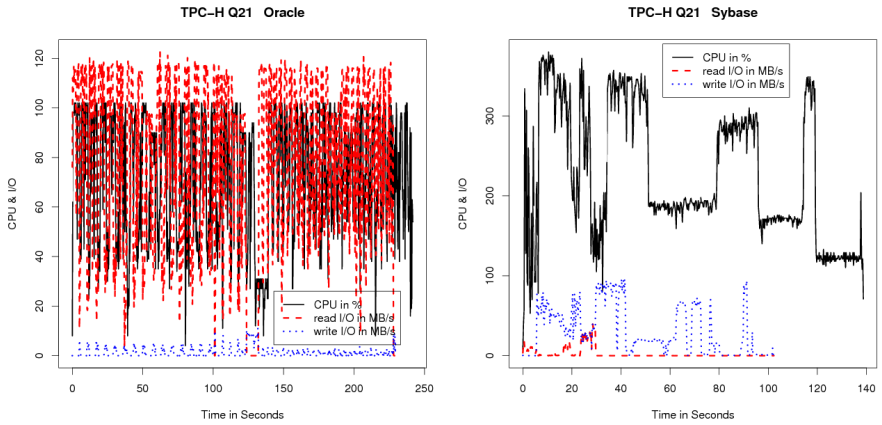


Figure A.52.: *CPU and I/O for TPC-H Q21 on Oracle and Sybase.*

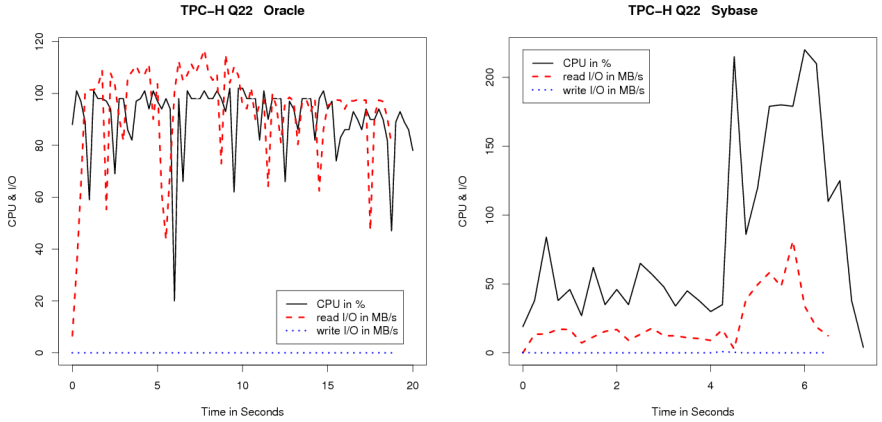


Figure A.53.: *CPU and I/O for TPC-H Q22 on Oracle and Sybase.*

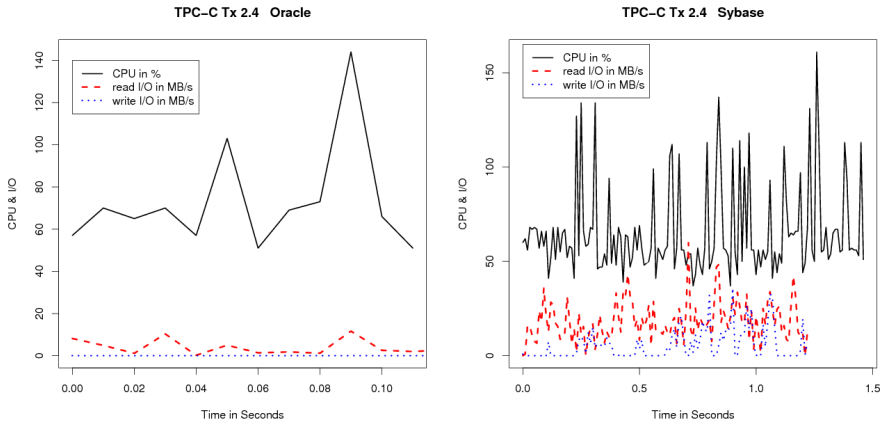


Figure A.54.: *CPU and I/O for TPC-C Transaction 2.4 on Oracle and Sybase.*

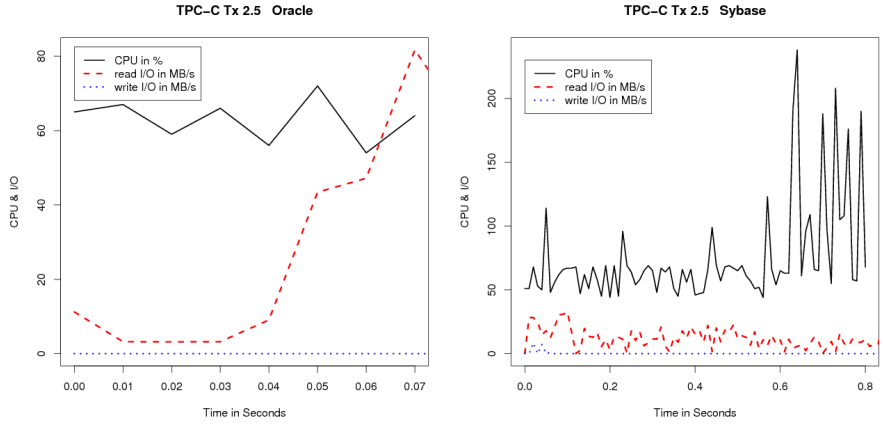


Figure A.55.: *CPU and I/O for TPC-C Transaction 2.5 on Oracle and Sybase.*

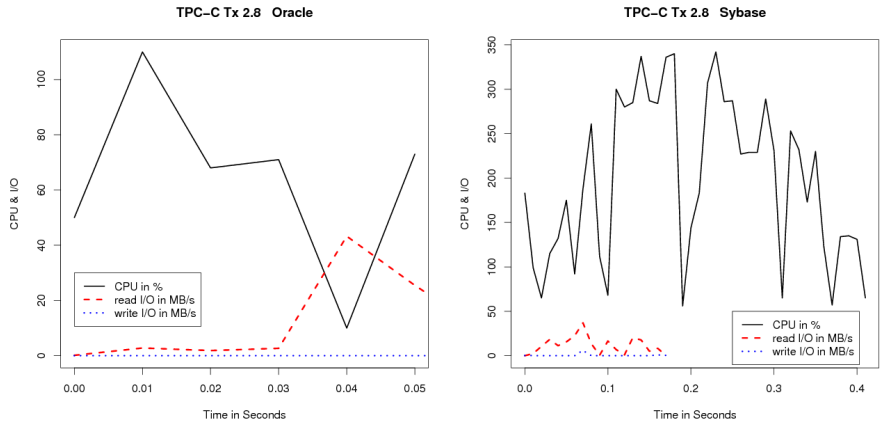


Figure A.56.: *CPU and I/O for TPC-C Transaction 2.8 on Oracle and Sybase.*

A.5. Additional Methods for Column-Store Implementation in HSQLDB

We present additional methods in this section, which we have derived and adapted from HSQLDB methods for our prototype. For functionality descriptions, we refer to Section 6.4.3.

```
1  import java.util.HashMap;
2  import org.hsqldb.rowio.RowOutputInterface;
3  import org.hsqldb.types.Type;
4
5  public class MyColumn {
6      public  HashMap<Integer, Object>  Components;
7      public  HashMap<Integer, Integer>  FilePositions;
8      private Type  type;
9
10     public MyColumn(HashMap<Integer, Object>comps, HashMap<Integer, Integer> pos, Type t) {
11         Components   = comps;
12         FilePositions   = pos;
13         type = t;
14     }
15     public MyColumn(Type t) {
16         Components   = new HashMap<Integer, Object>();
17         FilePositions   = new HashMap<Integer, Integer>();
18         type = t;
19     }
20     public void setType(Type t) {
21         type = t;
22     }
23     public void write(int index, RowOutputInterface out) {
24         if (Components.containsKey(index) == false) {return;}
25         Object myData = Components.get(index);
26         out.writeData(type, myData); out.writeEnd();
27     }
28     public void writeAll(RowOutputInterface out) {
29         for (int i : Components.keySet()) {write(i, out);}
30     }
31     /** @param index @param out @return length of Components[index] in Byte for the corresponding source
         file (incl. separator "|") */
32     public int getSize(int index, RowOutputInterface out) {
33         out.reset(); write(index, out);
34         int size = out.getOutputStream().size();
35         out.reset();
36         return size;
37     }
38 }
```

Figure A.57.: *Class MyColumn for columnar representation in HSQLDB.*

```
1  public CachedObject[] get(RowInputInterface[] ins) { // ins contains (a chunk of) a column
2      final int chunkSize = TextCache.CHUNKSIZE;
3      Type[] colsTypes = table.getColumnTypes();
4      Object[][] cols = new Object[table.getColumnCount()][chunkSize]; // 1st element = column; 2nd element
         = row
5      RowAVLDiskData[] rows = new RowAVLDiskData[chunkSize]; // currently constant 100 rows
6      int rowPos = ((RowInputText) ins[0]).rowPos;
7      for (int i = 0; i < ins.length; i++) { // COLUMNS
8          try {
9              for (int j = 0; j < chunkSize; j++) {
10                 cols[i][j] = ins[i].readData(colsTypes[i]);
11                 ((TextCache) this.cache).cols[i].Components.put(rowPos + j, cols[i][j]); // create MyColumns
12             }
13         } catch (IOException e) {throw Error.error(ErrorCode.TEXT_FILE_IO, e);}
14     }
15     // columns to rows --> build result
16     Object[][] data = new Object[chunkSize][ins.length];
17     for (int j = 0 ; j < chunkSize; j++) {
18         if (cols[0][j] == null)
19                 break; // aborts if < 100 values
20         for (int i = 0; i < cols.length; i++) {data[j][i] = cols[i][j];}
21             rows[j] = new RowAVLDiskData(this, table, data[j]);
22             rows[j].setPos(rowPos + j);
23             rows[j].setStorageSize(1);
24             rows[j].setChanged(false);
25     }
26     if (currentRow != null) {currentRow.setData(data[0]);}
27     return rows;
28     }
```

Figure A.58.: *New method get for column-wise organized data (class RowStoreAVLDiskData).*

```
1  protected CachedObject getFromFile(int rowPos, PersistentStore store, boolean keep) {
2          CachedObject object = null;
3          CachedObject[] rows = new CachedObject[TextCache.CHUNKSIZE];
4          writeLock.lock();
5          try {
6              object = cache.get(rowPos);
7              if (object != null) {
8                  if (keep) {object.keepInMemory(true);}
9                  return object; // from cache
10             } // if row requested from file, #CHUNKSIZE rows loaded --> next row in cache (sequentially)
11             for (int j = 0; j < 2; j++) {
12                 try { // read data from files
13                     RowInputInterface[] colInputs = myReadObject(rowPos); // chunk from (column) files
14                     if (colInputs[0] == null) {return null;}
15                     rows = ((RowStoreAVLDiskData) store).get(colInputs); //RowStoreAVLDiskdata.get()
16                     break;
17                 }
18                 catch (OutOfMemoryError err) {cache.forceCleanUp(); System.gc(); if (j > 0) {throw err;}}
19             }
20             for (int i = 0; i < rows.length; i++) {
21                 if (rows[i] == null) {break;}
22                 cache.put(rowPos + i, rows[i]); // add row to cache
23                 if (keep) {rows[i].keepInMemory(true);}
24             }
25             return rows[0]; // returns first element, which was requested
26         } catch (HsqlException e) {database.logger.logSevereEvent(dataFileName + " getFromFile " +
             rowPos, e); throw e;}
27         finally {writeLock.unlock();}
28     }
```

Figure A.59.: *Alternative get method for data not resident in cache (class TextCache).*

```
1  ...
2  if (Features.STORAGE_TYPE == Features.ROW_STORE) { // original implementation (row by row)
3      if (it.next()) { // it.next fetches next row which satisfies conditions
4              if (currentIndex < rangeVariables.length - 1) {currentIndex++; continue;}
5      } else {it.reset(); currentIndex--; continue;}
6  }
7  if (Features.STORAGE_TYPE == Features.COLUMN_STORE) {
8      if (it.myNext(cache, rowPos, count, currentIndex)) { // condition check and it.myNext fetches next
         column value (row of column)
9              if (currentIndex < rangeVariables.length - 1) {currentIndex++; continue;}
10             if (rowPos[currentIndex] > count[currentIndex]) {
11                     if (rowPos[0] >= count[0]) break; it.reset(); rowPos[currentIndex] = 1; currentIndex--;
12                     if (currentIndex >= 0) rowPos[currentIndex]++; continue;
13             }
14     } else {
15             if (rowPos[0] >= count[0] +1) break; it.reset(); rowPos[currentIndex] = 1; // set outer index to 0
16             currentIndex--;
17             if (currentIndex >= 0) rowPos[currentIndex]++; continue; // increase inner index
18     }
19 } ...
```

Figure A.60.: *Query dispatching for aggregates in Row Store and Column Store configuration (class QuerySpecification).*

```
1  ...
2  public boolean myNext( Object cache[], int[] position, int maxPos[], int currentIndex) {
3      while (condIndex < conditions.length) {
4          if (isBeforeFirst) {isBeforeFirst = false; initialiseIterator();}
5          boolean result = myFindNext(cache, position, maxPos, currentIndex);
6          if (result) {return true;}
7          reset(); condIndex++;
8      }
9      condIndex = 0;
10     return false;
11   } ...
12
13 protected boolean myFindNext(Object cache[], int[] position, int maxPos[], int currentIndex) { // fetches
     next row which fits the conditions
14     boolean result = false;
15     while (true) {
16         if (conditions[condIndex].indexEndCondition != null && !conditions[condIndex].indexEndCondition
             .myTestCondition(session, cache, position, currentIndex)) {
17             if (!conditions[condIndex].isJoin) {hasLeftOuterRow = false;}
18             break;
19         }
20         if (joinConditions[condIndex].nonIndexCondition != null &&
             !joinConditions[condIndex].nonIndexCondition .myTestCondition(session, cache, position,
             currentIndex)) {
21             if (position[currentIndex] < maxPos[currentIndex]) {
22                     position[currentIndex]++; continue;
23             } else {return false;}
24         }
25         if (whereConditions[condIndex].nonIndexCondition != null &&
             !whereConditions[condIndex].nonIndexCondition .myTestCondition(session, cache, position,
             currentIndex)) {
26             hasLeftOuterRow = false; addFoundRow(); continue;
27         }
28         if (position.length == 1 && position[0] >= maxPos[0]+1) {return false;}
29         Expression e = conditions[condIndex].excludeConditions;
30         if (e != null && e.testCondition(session)) {continue;}
31         addFoundRow();
32         hasLeftOuterRow = false;
33         return true;
34     }
35     it.release();
36     currentRow = null;
37     currentData = rangeVar.emptyData;
38     if (hasLeftOuterRow && condIndex == conditions.length - 1) {
39         result = (whereConditions[condIndex].nonIndexCondition == null ||
             whereConditions[condIndex].nonIndexCondition .myTestCondition(session, cache, position,
             currentIndex));
40         hasLeftOuterRow = false;
41     }
42     return result;
43   } ...
```

Figure A.61.: *Predicate evaluation methods for alternative data-flow path: myNext and myFindNext (class RangeVariable).*

Bibliography

[Aba08] Daniel J. Abadi. *Query execution in column-oriented database systems.* PhD thesis, Massachusetts Institute of Technology, Cambridge, USA, 2008. Adviser: Madden, Samuel R.

[ABC+76] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, and Vera Watson. System R: Relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[ABH09] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. Column oriented database systems. *The Proceedings of the VLDB Endowment (PVLDB)*, 2(2):1664–1665, 2009.

[ABH+13] Daniel J. Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel R. Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases (Found and Trends DB)*, 5(3):197–280, December 2013.

[ABKS13] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. *Feature-oriented software product lines - Concepts and implementation.* Springer, Heidelberg, Germany, 2013.

[ACK+04] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. Database tuning advisor for Microsoft SQL Server 2005. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1110–1121, San Francisco, USA, 2004. Morgan Kaufmann.

[ACN00] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 496–505, San Francisco, USA, 2000. Morgan Kaufmann.

[ACN06] Sanjay Agrawal, Eric Chu, and Vivek R. Narasayya. Automatic physical design tuning: Workload as a sequence. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 683–694, New York, USA, 2006. Association for Computing Machinery.

[ADHS01] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 169–180, San Francisco, USA, 2001. Morgan Kaufmann.

[AIA14] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. H2O: A hands-free adaptive store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1103–1114, New York, USA, 2014. Association for Computing Machinery.

[AMDM07] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel R. Madden. Materialization strategies in a column-oriented DBMS. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 466–475, Washington D.C., USA, 2007. IEEE Computer Society.

[Ame92] American National Standards Institute. *American national standard for information systems: Database language — SQL: ANSI X3.135-1992*. American National Standards Institute, New York, USA, 1992. Revision and consolidation of ANSI X3.135-1989 and ANSI X3.168-1989, Approved October 3, 1989.

[AMF06] Daniel J. Abadi, Samuel R. Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 671–682, New York, USA, 2006. Association for Computing Machinery.

[AMH08] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: How different are they really? In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 967–980, New York, USA, 2008. Association for Computing Machinery.

[APM16] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 583–598, New York, USA, 2016. Association for Computing Machinery.

[Bat79] Don S. Batory. On searching transposed files. *ACM Transactions on Database Systems (TODS)*, 4(4):531–544, December 1979.

[BBJB14] Selma Bouarar, Ladjel Bellatreche, Stéphane Jean, and Mickaël Baron. Do rule-based approaches still make sense in logical data warehouse design? In *Proceedings of the East European Conference on Advances in Databases and Information Systems (ADBIS)*, volume 8716 of *Lecture Notes in Computer Science*, pages 83–96, Heidelberg, Germany, 2014. Springer.

[BBO16] Lahcène Brahimi, Ladjel Bellatreche, and Yassine Ouhammou. A recommender system for DBMS selection based on a test data repository. In *Proceedings of the East European Conference on Advances in Databases and Information Systems (ADBIS)*, volume 9809 of *Lecture Notes in Computer Science*, pages 166–180, Heidelberg, Germany, 2016. Springer.

[BC05] Nicolas Bruno and Surajit Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 227–238, New York, USA, 2005. Association for Computing Machinery.

[BC06] Nicolas Bruno and Surajit Chaudhuri. To tune or not to tune? A lightweight physical design alerter. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 499–510, New York, USA, 2006. Association for Computing Machinery.

[BC07] Nicolas Bruno and Surajit Chaudhuri. An online approach to physical design tuning. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 826–835, Washington D.C., USA, 2007. IEEE Computer Society.

[BCB10] Ladjel Bellatreche, Alfredo Cuzzocrea, and Soumia Benkrid. Query optimization over parallel relational data warehouses in distributed environments by simultaneous fragmentation and allocation. In *Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing, Part I (ICA3PP (1))*, volume 6081 of *Lecture Notes in Computer Science*, pages 124–135, Heidelberg, Germany, 2010. Springer.

[BDD+98] Randall G. Bello, Karl Dias, Alan Downing, James J. Feenan Jr., James L. Finnerty, William D. Norcott, Harry Sun, Andrew Witkowski, and Mohamed Ziauddin. Materialized views in Oracle. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 659–664, San Francisco, USA, 1998. Morgan Kaufmann.

[BFB14] Ahcène Boukorca, Zoé Faget, and Ladjel Bellatreche. What-if physical design for multiple query plan generation. In *Proceedings of the International Conference on Database and Expert Systems Applications, Part I (DEXA (I))*, volume 8644 of *Lecture Notes in Computer Science*, pages 492–506, Heidelberg, Germany, 2014. Springer.

[BK10] Bettina Berendt and Veit Köppen. Improving ranking by respecting the multidimensionality and uncertainty of user preferences. In *Intelligent Information Access (SCI vol. 301)*, volume 301 of *Studies in Computational Intelligence*, pages 39–56. Springer, Heidelberg, Germany, 2010.

[BM70] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control (SIGFIDET (SIGMOD))*, pages 107–141, New York, USA, 1970. Association for Computing Machinery.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica (Acta Inf.)*, 1(3):173–189, 1972.

[BM00] Zohra Bellahsene and P. Marot. Materializing a set of views: Dynamic strategies and performance evaluation. In *Proceedings of the International Database Engineering & Applications Symposium (IDEAS)*, pages 424–430, Washington D.C., USA, 2000. IEEE Computer Society.

[Böß09] Daniel Bößwetter. SPAX - PAX with super-pages. In *Proceedings of the East European Conference on Advances in Databases and Information Systems (ADBIS)*, volume 5739 of *Lecture Notes in Computer Science*, pages 362–377, Heidelberg, Germany, 2009. Springer.

[BR91] Danail G. Bonchev and Dennis H. Rouvray, editors. *Chemical graph theory: Introduction and fundamentals*. Gordon and Breach Science Publishers, New York, USA, 1st edition, 1991.

[Bra12] Darius Braziūnas. *Decision-theoretic elicitation of additive utilities*. PhD thesis, University of Toronto, Toronto, Canada, 2012. Adviser: Boutilier, Craig.

[Bru11] Nicolas Bruno. *Automated Physical Database Design and Tuning*. Emerging directions in database systems and applications. CRC Press, Taylor & Francis Group, Boca Raton, USA, 2011.

[Bub07] Ronny Bubke. Dynamische Ermittlung und Verwaltung von materialisierten Sichten auf Grundlage des Query Graph Models. Master thesis (Diplomarbeit), School of Computer Science, Magdeburg, Germany, 2007. In German; Adviser: Eike Schallehn.

[Cab70] A. Victor Cabot. An enumeration algorithm for knapsack problems. *Journal of the Operations Research (Operations Research)*, 18(2):306–311, 1970.

[CAB+81] Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond A. Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia P. Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. A history and evaluation of System R. *Communications of the ACM (CACM)*, 24(10):632–646, 1981.

[CAE+76] Donald D. Chamberlin, Morton M. Astrahan, Kapali P. Eswaran, Patricia P. Griffiths, Raymond A. Lorie, James W. Mehl, Phyllis Reisner, and Bradford W. Wade. SEQUEL 2: A unified approach to data definition, manipulation, and control. *IBM Journal of Research and Development (IBM J. Res. Dev.)*, 20(6):560–575, November 1976.

[CB74] Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A structured English query language. In *Proceedings of the ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control (SIGFIDET (SIGMOD))*, pages 249–264, New York, USA, 1974. Association for Computing Machinery.

[CBC93a] Sunil Choenni, Henk M. Blanken, and Thiel Chang. Index selection in relational databases. In *Proceeding of the International Conference on Computing and Information (ICCI)*, pages 491–496, Washington D.C., USA, 1993. IEEE Computer Society.

[CBC93b] Sunil Choenni, Henk M. Blanken, and Thiel Chang. On the selection of secondary indices in relational databases. *Data Knowledge Engineering (DKE)*, 11(3):207–233, 1993.

[CCS93] Edgar F. Codd, Sally B. Codd, and Clynch T. Salley. Providing OLAP to user-analysts: An IT mandate. White Paper, 1993.

[CDF+01] Eugene Inseok Chong, Souripriya Das, Chuck Freiwald, Jagannathan Srinivasan, Aravind Yalamanchi, Mahesh Jagannath, Anh-Tuan Tran, and Ramkumar Krishnan. B+-tree indexes with hybrid row identifiers in Oracle 8i. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 341–348, Washington D.C., USA, 2001. IEEE Computer Society.

[CDN04] Surajit Chaudhuri, Mayur Datar, and Vivek R. Narasayya. Index selection for databases: A hardness study and a principled heuristic solution. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(11):1313–1323, 2004.

[CFG+11] Richard L. Cole, Florian A. Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. The mixed workload ch-benchmark. In *Proceedings of the International Workshop on Testing Database Systems (DBTest)*, pages 1–6, New York, USA, 2011. Association for Computing Machinery.

[CFM95] Alberto Caprara, Matteo Fischetti, and Dario Maio. Exact and approximate algorithms for the index selection problem in physical database design. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 7(6):955–967, 1995.

[CG85] Stefano Ceri and Georg Gottlob. Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries. *IEEE Transactions on Software Engineering (TSE)*, 11:324–345, 1985.

[CG94] Richard L. Cole and Goetz Graefe. Optimization of dynamic query evaluation plans. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 150–160, New York, USA, 1994. Association for Computing Machinery.

[CGN02] Surajit Chaudhuri, Ashish K. Gupta, and Vivek R. Narasayya. Compressing SQL workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 488–499, New York, USA, 2002. Association for Computing Machinery.

[Cha98] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 34–43, New York, USA, 1998. Association for Computing Machinery.

[Che76] Peter Pin-Shan Chen. The entity-relationship model - Toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.

[Chi68a] David L. Childs. Description of a set-theoretic data structure. In *American Federation of Information Processing Societies: Proceedings of the Fall Joint Computer Conference, Part I (AFIPS (Fall, Part I))*, pages 557–564, New York, USA, 1968. Association for Computing Machinery.

[Chi68b] David. L. Childs. Feasibility of a set-theoretic data structure. A general structure based on a reconstituted definition of relation. In *International Federation for Information Processing World Computer Congress (Part 1) (IFIP (1))*, pages 420–430, Laxenburg, Austria, 1968. International Federation for Information Processing.

[Chi68c] David. L. Childs. Feasibility of a set-theoretic data structure. A general structure based on a reconstituted definition of relation. Technical Report Nr. 6, University of Michigan, Ann Arbor, USA, 1968.

[Chv83] Vasek Chvatal. *Linear programming.* W. H. Freeman and Company, 1st edition, 1983.

[CI98] Chee-Yong Chan and Yannis E. Ioannidis. Bitmap index design and evaluation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 355–366, New York, USA, 1998. Association for Computing Machinery.

[CK85] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 268–279, New York, USA, 1985. Association for Computing Machinery.

[CKPS95] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseak Shim. Optimizing queries with materialized views. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 190–200, Washington D.C., USA, 1995. IEEE Computer Society.

[CKR05] Surajit Chaudhuri, Raghav Kaushik, and Ravishankar Ramamurthy. When can we trust progress estimators for SQL queries? In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 575–586, New York, USA, 2005. Association for Computing Machinery.

[CN97] Surajit Chaudhuri and Vivek R. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL Server. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 146–155, New York, USA, 1997. Association for Computing Machinery.

[CN98] Surajit Chaudhuri and Vivek R. Narasayya. AutoAdmin "what-if" index analysis utility. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 367–378, New York, USA, 1998. Association for Computing Machinery.

[CN07] Surajit Chaudhuri and Vivek R. Narasayya. Self-tuning database systems: A decade of progress. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 3–14, New York, USA, 2007. Association for Computing Machinery.

[Cod70] Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM (CACM)*, 13(6):377–387, 1970.

[Cod72] Edgar F. Codd. Relational completeness of data base sublanguages. Technical Report IBM Research Report RJ 987 #17041, IBM Research, San Jose, USA, 1972.

[Com78] Douglas Comer. The difficulty of optimum index selection. *ACM Transactions on Database Systems (TODS)*, 3(4):440–445, 1978.

[Com79] Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, June 1979.

[CS95] Surajit Chaudhuri and Kyuseok Shim. An overview of cost-based optimization of queries with aggregates. *IEEE Data Engineering Bulletin (Data Eng. Bull.)*, 18(3):3–9, 1995.

[CY90] Douglas W. Cornell and Philip S. Yu. An effective approach to vertical partitioning for physical design of relational databases. *IEEE Transactions on Software Engineering (TSE)*, 16(2):248–258, 1990.

[DD97] Chris J. Date and Hugh Darwen. *A guide to SQL standard*. Pearson/Addison-Wesley, Boston, USA, 4th edition, 1997.

[DDD+04] Benoit Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zait, and Mohamed Ziauddin. Automatic SQL tuning in Oracle 10g. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1098–1109, San Francisco, USA, 2004. VLDB Endowment.

[Des07] P. S. Deshpande. *SQL & PL/SQL for Oracle 10g black book*. Dreamtech Press, New Dehli, India, reprint 2009 edition, 2007.

[DG85] David J. DeWitt and Robert H. Gerber. Multiprocessor hash-based join algorithms. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 151–164, San Francisco, USA, 1985. Morgan Kaufmann.

[DL65] D. R. Davis and A. D. Lin. Secondary key retrieval using an IBM 7090-1301 system. *Communications of the ACM (CACM)*, 8(4):243–246, April 1965.

[ED95] Cem Evrendilek and Asuman Dogac. Query decomposition, optimization and processing in multidatabase systems. In *Proceedings of the International Workshop on Next Generation Information Technologies and Systems (NGITS)*, page N.N., Haifa, Israel, 1995. Technion - Israel Institute of Technology.

[EM99] Andrew Eisenberg and Jim Melton. SQL: 1999, formerly known as SQL 3. *SIGMOD Record (SIGMOD Rec.)*, 28(1):131–138, 1999.

[EN10] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of database systems*. Pearson/Addison-Wesley, Boston, USA, 6th edition, 2010.

[ENR09] Hicham G. Elmongui, Vivek R. Narasayya, and Ravishankar Ramamurthy. A framework for testing query transformation rules. In *Proceedings of the SIGMOD International Conference on Managementof Data (SIGMOD)*, pages 257–268, New York, USA, 2009. Association for Computing Machinery.

[FBB07] Cécile Favre, Fadila Bentayeb, and Omar Boussaid. Evolution of data warehouses' optimization: A workload perspective. In *Proceedings of the International Conference on Data Warehousing and Knowledge Discovery (DaWaK)*, number 4652 in Lecture Notes in Computer Science, pages 13–22, Heidelberg, Germany, 2007. Springer.

[FBK+16] Jan Finis, Robert Brunel, Alfons Kemper, Thomas Neumann, Norman May, and Franz Färber. Order indexes: Supporting highly dynamic hierarchical data in relational main-memory database systems. *The VLDB Journal (VLDB J.)*, Special issue:1–26, August 2016.

[Fer06] Diego R. Llanos Ferraris. TPCC-UVa: An open-source TPC-C implementation for global performance measurement of computer systems. *SIGMOD Record (SIGMOD Rec.)*, 35(4):6–15, 2006.

[FGE05] José Figueira, Salvatore Greco, and Matthias Ehrgott, editors. *Multiple criteria decision analysis: State of the art surveys.* Springer, Heidelberg, Germany, 1st edition, 2005.

[FGK02] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: A modeling language for mathematical programming.* Duxbury Press, 2nd edition, 2002.

[Fis67a] Peter C. Fishburn. Interdependence and additivity in multivariate, unidimensional expected utility theory. *International Economic Review (IER)*, 8(3):335–342, 1967.

[Fis67b] Peter C. Fishburn. Methods of estimating additive utilities. *Management Science (ManSci)*, 13(7):435–453, 1967.

[FKN12] Florian A. Funke, Alfons Kemper, and Thomas Neumann. Compacting transactional data in hybrid OLTP&OLAP databases. *The Proceedings of the VLDB Endowment (PVLDB)*, 5(11):1424–1435, July 2012.

[FLP+08] Camille Furtado, Alexandre A. B. Lima, Esther Pacitti, Patrick Valduriez, and Marta Mattoso. Adaptive hybrid partitioning for OLAP query processing in a database cluster. *International Journal of High Performance Computing and Networking (IJHPCN)*, 5(4):251–262, 2008.

[FM02] Paul J. Fortier and Howard Michel. *Computer systems performance evaluation and prediction.* Butterworth-Heinemann, Newton, USA, 2002.

[Fre87] Johann Christoph Freytag. A rule-based view of query optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, volume 16 (3), pages 173–180, New York, USA, 1987. Association for Computing Machinery.

[FST88] Sheldon J. Finkelstein, Mario Schkolnick, and Paolo Tiberio. Physical database design for relational databases. *ACM Transactions on Database Systems (TODS)*, 13(1):91–128, 1988.

[Fun15] Florian A. Funke. *Adaptive physical optimization in hybrid OLTP&OLAP main-memory database systems.* PhD thesis, Technical University of Munich, Munich, Germany, 2015. Adviser: Kemper, Alfons.

[GBLP95] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. Technical Report MSR-TR-95-22, Advanced Technology Division, Microsoft Research, Redmond, USA, 1995.

[GCB+97] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery (Data Min. Knowl. Discov.)*, 1(1):29–53, 1997.

[GCCK07] Daniel Gmach, Jerry R. Cherkasova, Ludmila Cherkasova, and Alfons Kemper. Workload analysis and demand prediction of enterprise data center applications. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 171–180, Washington D.C., USA, 2007. IEEE Computer Society.

[GD87] Goetz Graefe and David J. DeWitt. The EXODUS optimizer generator. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 160–172, New York, USA, 1987. Association for Computing Machinery.

[GHQ95] Ashish Gupta, Venky Harinarayan, and Dallan Quass. Aggregate-query processing in data warehousing environments. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 358–369, San Francisco, USA, 1995. Morgan Kaufmann.

[GHRU97] Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Index selection for OLAP. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 208–219, Washington D.C., USA, 1997. IEEE Computer Society.

[GK10] Goetz Graefe and Harumi Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 371–381, New York, USA, 2010. Association for Computing Machinery.

[GKP+10] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel R. Madden. HYRISE: A main memory hybrid storage engine. *The Proceedings of the VLDB Endowment (PVLDB)*, 4(2):105–116, November 2010.

[GLS94] Goetz Graefe, Amanda Linville, and Leonard D. Shapiro. Sort vs. hash revisited. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 6(6):934–944, December 1994.

[GM93] Goetz Graefe and William J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 209–218, Washington D.C., USA, 1993. IEEE Computer Society.

[GNU10] GNU Operating System. Bash reference manual. Available online, 2010. Used version 4.1.5, `http://www.gnu.org/software/bash/manual/bashref.html` (viewed August 2016).

[GP99] Peter Gulutzan and Trudy Pelzer. *SQL-99 complete, really.* CMP Books, Lawrence, USA, 1st edition, 1999.

[GPSH02] Antara Ghosh, Jignashu Parikh, Vibhuti S. Sengar, and Jayant R. Haritsa. Plan selection based on query clustering. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 179–190, San Francisco, USA, 2002. Morgan Kaufmann.

[Gra90] Goetz Graefe. Volcano, An extensible and parallel query evaluation system. Technical Report CU-CS-481-90, Department of Computer Science, University of Colorado, Boulder, USA, 1990.

[Gra94a] Goetz Graefe. Sort-merge-join: An idea whose time has(h) passed? In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 406–417, Washington D.C., USA, 1994. IEEE Computer Society.

[Gra94b] Goetz Graefe. Volcano — An extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 6(1):120–135, February 1994.

[Gra00] Goetz Graefe. Dynamic query evaluation plans: Some course corrections? *IEEE Data Engineering Bulletin (Data Eng. Bull.)*, 23(2):3–6, June 2000.

[Han10] Eric N. Hanson. Columnstore indexes for fast data warehouse query processing in SQL Server 11.0. White Paper, 2010.

[HC76] Michael Hammer and Arvola Chan. Index selection in a self-adaptive data base management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1–8, New York, USA, 1976. Association for Computing Machinery.

[HD08] Allison L. Holloway and David J. DeWitt. Read-optimized databases, in depth. *The Proceedings of the VLDB Endowment (PVLDB)*, 1(1):502–513, August 2008.

[Hen00] Ken Henderson. *The guru's guide to Transact-SQL.* Pearson/Addison-Wesley, Menlo Park, USA, 1st edition, 2000.

[HGR09] Marc Holze, Claas Gaidies, and Norbert Ritter. Consistent on-line classification of DBS workload events. In *Proceedings of the ACM Conference on Information and Knowledge Management (CIKM)*, pages 1641–1644, New York, USA, 2009. Association for Computing Machinery.

[HIKY12] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *The Proceedings of the VLDB Endowment (PVLDB)*, 5(6):502–513, 2012.

[HJP03] Cinda Heeren, Hosagrahar V. Jagadish, and Leonard Pitt. Optimal indexing using near-minimal space. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 244–251, New York, USA, 2003. Association for Computing Machinery.

[HLAM06] Stavros Harizopoulos, Velen Liang, Daniel J. Abadi, and Samuel R. Madden. Performance tradeoffs in read-optimized databases. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 487–498, New York, USA, 2006. Association for Computing Machinery.

[How10] Philip Howard. Sybase IQ 15.2 – An InDetail paper. White Paper, available online, 2010. `https://www.sybase.ru/dl/files/IQ/Bloor_Sybase_IQ_15.2.pdf` (viewed May 2016).

[HP03] Richard A. Hankins and Jignesh M. Patel. Data morphing: An adaptive, cache-conscious storage technique. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 417–428, San Francisco, USA, 2003. Morgan Kaufmann.

[HP12] John L. Hennessy and David A. Patterson. *Computer architecture: A quantitative approach*. Morgan Kaufmann, Waltham, USA, 5th edition, 2012.

[HR83] Theo Härder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, December 1983.

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning*. Springer, Heidelberg, Germany, 2nd edition, 2009.

[IBM05] IBM Corporation. An architectural blueprint for autonomic computing. White Paper, 2005. 3rd edition.

[IBM06a] IBM Corporation. An architectural blueprint for autonomic computing. White Paper, 2006. 4th edition.

[IBM06b] IBM Corporation. DB2 Version 9 — Performance guide. White Paper, 2006. SC10- 4222-00.

[IBM06c] IBM Corporation. Introducing DB2 9, part 4: Automatic and other enhancements in DB2 9. Technical Article, 2006.

[Idr10] Stratos Idreos. *Database cracking: Towards auto-tuning database kernels.* PhD thesis, Centrum Wiskunde & Informatica, Amsterdam, Netherlands, 2010. Adviser: Martin L. Kersten.

[IKM07a] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database cracking. In *Proceedings of the International Conference on Innovative Data Systems Research (CIDR)*, pages 68–78, Madison, USA, 2007. www.cidrdb.org.

[IKM07b] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Updating a cracked database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 413–424, New York, USA, 2007. Association for Computing Machinery.

[Inf08] Infobright Inc. Infobright community edition. White Paper, 2008. ICE 3.2.2 documentation pack.

[Inf11a] Infobright Inc. Infobright analytic database. White Paper, 2011. Architecture overview.

[Inf11b] Infobright Inc. Infobright analytic database. White Paper, 2011. Technical brief.

[Ing09] Ingres. Ingres/VectorWise sneak preview on the Intel Xeon processor 5500 series-based platform. White Paper, 2009.

[Inm05] William H. Inmon. *Building the data warehouse.* Wiley, Hoboken, USA, 4th edition, 2005.

[Int92] International Organization for Standardization. *ISO/IEC 9075:1992: Information technology – Database languages – SQL.* International Organization for Standardization, Geneva, Switzerland, 1992. Revision and consolidation of ISO/IEC 9075:1989.

[Int99] International Organization for Standardization. *ISO/IEC 9075-X:1999: Information technology – Database languages – SQL – Part 1-5.* International Organization for Standardization, Geneva, Switzerland, 1999. Part 1: Framework (SQL/Framework), Part 2: Foundation (SQL/Foundation), Part 3: Call-Level Interface (SQL/CLI), Part 4: Persistent Stored Modules (SQL/PSM), Part 5: Host Language Bindings, Amd 1:2001: On-Line Analytical Processing (SQL/OLAP).

[Int03a] International Organization for Standardization. *ISO/IEC 9075-14:2003: Information technology – Database languages – SQL – Part 13: SQL routines and types using the JAVA® programming language.* International Organization for Standardization, Geneva, Switzerland, 2003.

[Int03b] International Organization for Standardization. *ISO/IEC 9075-14:2003: Information technology – Database languages – SQL – Part 14: XML-related specifications (SQL/XML)*. International Organization for Standardization, Geneva, Switzerland, 2003.

[Ioa96] Yannis E. Ioannidis. Query optimization. *ACM Computing Surveys (CSUR)*, 28(1):121–123, 1996.

[ISR83] Maggie Y.L. Ip, Lawrence V. Saxton, and Vijay V. Raghavan. On the selection of an optimal set of indexes. *IEEE Transactions on Software Engineering (TSE)*, 9(2):135–143, 1983.

[JK84] Matthias Jarke and Jürgen Koch. Query optimization in database systems. *ACM Computing Surveys (CSUR)*, 16(2):111–152, 1984.

[Joh61] L. R. Johnson. An indirect chaining method for addressing on secondary keys. *Communications of the ACM (CACM)*, 4(5):218–222, May 1961.

[KBL05] Michael Kifer, Arthur J. Bernstein, and Philip M. Lewis. *Database systems: An application-oriented approach (complete version)*. Pearson/Addison-Wesley, London, UK, 2nd edition, 2005.

[KCJ+87] Setrag N. Khoshafian, George P. Copeland, Thomas Jagodits, Haran Boral, and Patrick Valduriez. A query processing strategy for the decomposed storage model. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 636–643, Washington D.C., USA, February 1987. IEEE Computer Society.

[KKG+11] Jens Krueger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. Fast updates on read-optimized databases using multi-core CPUs. *The Proceedings of the VLDB Endowment (PVLDB)*, 5(1):61–72, September 2011.

[KLS+03] Eva Kwan, Sam Lightstone, K. Bernhard Schiefer, Adam J. Storm, and Leanne Wu. Automatic database configuration for DB2 Universal Database: Compressing years of performance expertise into seconds of execution. In *Proceedings of the GI-conference on Database Systems in Business, Technology and Web (BTW)*, pages 620–629, Bonn, Germany, 2003. German Informatics Society.

[KN10] Alfons Kemper and Thomas Neumann. One size fits all, again! The architecture of the hybrid OLTP & OLAP database management system HyPer. In *Proceedings of the International VLDB Workshop on Business Intelligence for the Real-Time Enterprise (BIRTE)*, volume 84 of *LNBIP (2011)*, pages 7–23, Heidelberg, Germany, 2010. Springer.

[KN11] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 195–206, Washington D.C., USA, 2011. IEEE Computer Society.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Pearson/Addison-Wesley, Boston, USA, 1st edition, 1973.

[KPP04] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack problems*. Springer, Heidelberg, Germany, 1st edition, 2004.

[KR76] Ralph L. Keeney and Howard Raiffa. *Decisions with multiple objectives: Preferences and value tradeoffs*. Wiley, New York, USA, 1976.

[KS97] Henry F. Korth and Abraham Silberschatz. Database research faces the information explosion. *Communications of the ACM (CACM)*, 40(2):139–142, February 1997.

[KY83] Yahiko Kambayashi and Masatoshi Yoshikawa. Query processing utilizing dependencies and horizontal decomposition. *ACM SIGMOD Record (SIGMOD Rec.)*, 13(4):55–67, May 1983.

[Lan04] Justin Langseth. Real-time data warehousing: Challenges and solutions. White Paper, available online, 2004. `http://dssresources.com/papers/features/langseth/langseth02082004.html` (viewed January 2016).

[LBH+15] Per-Åke Larson, Adrian Birka, Eric N. Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. Real-time analytical processing with SQL Server. *The Proceedings of the VLDB Endowment (PVLDB)*, 8(12):1740–1751, August 2015.

[LGB08] Andreas Lübcke, Ingolf Geist, and Ronny Bubke. Dynamic detection and administration of materialized views based on the query graph model. In *Proceedings of the IEEE International Conference on Digital Information Management (ICDIM)*, pages 940–942, Washington D.C., USA, 2008. IEEE Computer Society.

[LGB09] Andreas Lübcke, Ingolf Geist, and Ronny Bubke. Dynamic construction and administration of the workload graph for materialized views selection. *International Journal of Information Studies (IJIS)*, 1(3):172–181, 2009.

[LKS10] Andreas Lübcke, Veit Köppen, and Gunter Saake. Towards selection of optimal storage architecture for relational databases. Technical Report Nr. 11, School of Computer Science, University of Magdeburg, Magdeburg, Germany, 2010.

[LKS11a] Andreas Lübcke, Veit Köppen, and Gunter Saake. A decision model to select the optimal storage architecture for relational databases. In *Proceedings of the IEEE International Conference on Research Challenges in Information Science (RCIS)*, pages 74–84, Washington D.C., USA, 2011. IEEE Computer Society.

[LKS11b] Andreas Lübcke, Veit Köppen, and Gunter Saake. A query decomposition approach for relational DBMS using different storage architectures. Technical Report Nr. 11, School of Computer Science, University of Magdeburg, Magdeburg, Germany, 2011.

[LKS11c] Andreas Lübcke, Veit Köppen, and Gunter Saake. Workload representation across different storage architectures for relational DBMS. In *Proceedings of the GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken) (GvD)*, pages 79–84, Innsbruck, Austria, 2011. University of Innsbruck.

[LKS12] Andreas Lübcke, Veit Köppen, and Gunter Saake. Heuristics-based workload analysis for relational DBMSs. In *Proceedings of the International United Information Systems Conference (UNISCON)*, number 137 in Lecture Notes in Business Information Processing, pages 25–36, Heidelberg, Germany, 2012. Springer.

[LLR06] Thomas Legler, Wolfgang Lehner, and Andrew Ross. Data mining with the SAP NetWeaver BI Accelerator. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1059–1068. VLDB Endowment, 2006.

[LM67] Roger E. Levien and Melvin E. Maron. A computer system for inference execution and data retrieval. *Communications of the ACM (CACM)*, 10(11):715–721, November 1967.

[LMV04] Alexandre A. B. Lima, Marta Mattoso, and Patrick Valduriez. Adaptive virtual partitioning for OLAP query processing in a database cluster. In *Proceedings of the Brazilian Symposium on Databases (SBBD)*, pages 92–105, Brasilia, Brazil, 2004. University of Brasilia.

[LRBB04] Tilman Lange, Volker Roth, Mikio L. Braun, and Joachim M. Buhmann. Stability-based validation of clustering solutions. *Neural Computation (Neural Comput.)*, 16(6):1299–1323, June 2004.

[LS10] Andreas Lübcke and Gunter Saake. A framework for optimal selection of a storage architecture in RDBMS. In *Proceedings of the Baltic Conference on Databases & Information Systems (DB&IS)*, pages 65–76, Riga, Latvia, 2010. University of Latvia.

[LSKS12] Andreas Lübcke, Martin Schäler, Veit Köppen, and Gunter Saake. Workload-based heuristics for evaluation of physical database architectures. In *Proceedings of the Baltic Conference on Databases & Information Systems (DB&IS)*, pages 3–10, Vilnius, Lithuania, 2012. University of Vilnius.

[LSKS14] Andreas Lübcke, Martin Schäler, Veit Köppen, and Gunter Saake. Relational on demand data management for IT-services. In *Proceedings of the IEEE International Conference on Research Challenges in Information Science (RCIS)*, pages 561–572, Washington D.C., USA, 2014. IEEE Computer Society.

[LSS13] Andreas Lübcke, Martin Schäler, and Gunter Saake. Dynamic relational data management for technical applications. Technical Report Nr. 2, School of Computer Science, University of Magdeburg, Magdeburg, Germany, 2013.

[LSSS07a] Martin Lühring, Kai-Uwe Sattler, Eike Schallehn, and Karsten Schmidt. Autonomes Index Tuning - DBMS-integrierte Verwaltung von Soft-Indexen. In *Proceedings of the GI-conference on Database Systems in Business, Technology and Web (BTW)*, pages 152–171, Bonn, Germany, 2007. German Informatics Society.

[LSSS07b] Martin Lühring, Kai-Uwe Sattler, Karsten Schmidt, and Eike Schallehn. Autonomous management of soft indexes. In *Proceedings of the International Conference on Data Engineering Workshops (ICDE Workshops)*, pages 450–458, Washington D.C., USA, 2007. IEEE Computer Society.

[Lüb07] Andreas Lübcke. Self-Tuning für Bitmap-Index-Konfigurationen. In *Proceedings of the Studierendenprogramm at the GI-conference on Database Systems in Business, Technology and Web (BTW)*, pages 28–30, Bruchsal, Germany, 2007. International University.

[Lüb08] Andreas Lübcke. Cost-effective usage of bitmap-indexes in DS-systems. In *Proceedings of the GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken) (GvD)*, pages 96–100, Bruchsal, Germany, 2008. School of Information Technology, International University in Germany.

[Lüb09] Andreas Lübcke. Self-tuning of data allocation and storage management: Advantages and implications. In *Proceedings of the GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken) (GvD)*, pages 21–25, Rostock, Germany, 2009. University of Rostock.

[Lüb10] Andreas Lübcke. Challenges in workload analyses for column and row stores. In *Proceedings of the GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken) (GvD)*, volume 581, Aachen, Germany, 2010. CEUR-WS.org.

[Lum70] Vincent Y. Lum. Multi-attribute retrieval with combined indexes. *Communications of the ACM (CACM)*, 13(11):660–665, November 1970.

[MA04] R. Timothy Marler and Jasbir S. Arora. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization (Struct Multidiscipl Optim.)*, 26(6):369–395, 2004.

[MBNK04] Stefan Manegold, Peter A. Boncz, Niels Nes, and Martin L. Kersten. Cache-conscious radix-decluster projections. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 684–695, San Francisco, USA, 2004. Morgan Kaufmann.

[ME92] Priti Mishra and Margaret H. Eich. Join processing in relational databases. *ACM Computing Surveys (CSUR)*, 24(1):63–113, March 1992.

[Mey92] Stuart L. Meyer. *Data Analysis for Scientists and Engineers*. Peer Management Consultants Ltd., Evanston, USA, 2nd edition, 1992. Paperback Edition.

[MF04] Roger MacNicol and Blaine French. Sybase IQ Multiplex - Designed for analytics. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1227–1230. VLDB Endowment, 2004.

[MKKI13] Daniel Martin, Oliver Koeth, Johannes Kern, and Iliyana Ivanova. Near real-time analytics with IBM DB2 Analytics Accelerator. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 579–588. Association for Computing Machinery, 2013.

[MQM97] Inderpal Singh Mumick, Dallan Quass, and Barinderpal Singh Mumick. Maintenance of data cubes and summary tables in a warehouse. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 100–111, New York, USA, 1997. Association for Computing Machinery.

[MS77] Salvatore T. March and Dennis G. Serverance. The determination of efficient record segmentations and blocking factors for shared data files. *ACM Transactions on Database Systems (TODS)*, 2(3):279–296, September 1977.

[MS84] Salvatore T. March and Gary D. Scudder. On the selection of efficient record segmentations and backup strategies for large shared databases. *ACM Transactions on Database Systems (TODS)*, 9(3):409–438, September 1984.

[MS92] Jim Melton and Alan R. Simon. *Understanding the new SQL: A complete guide*. Morgan Kaufmann, San Francisco, USA, 1st edition, 1992.

[Mun57] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics (SIAM)*, 5(1):32–38, 1957.

[MZHW93] Axel Mönkeberg, Peter Zabback, Christof Hasse, and Gerhard Weikum. The COMFORT prototype: A step towards automated database performance tuning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 542–543, New York, USA, 1993. Association for Computing Machinery.

[NCWD84] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems (TODS)*, 9(4):680–710, December 1984.

[O'N87] Patrick E. O'Neil. MODEL 204 architecture and performance. In *Proceedings of the International Workshop on High Performance Transaction Systems (HTPS)*, number 359 in Lecture Notes in Computer Science, pages 39–59, Heidelberg, Germany, 1987. Springer.

[Ora03a] Oracle Corporation. Automatic storage management: Technical overview. White Paper, 2003.

[Ora03b] Oracle Corporation. Performance tuning using SQL Access Advisor. White Paper, 2003.

[Ora07] Oracle Corporation. Oracle database11g automatic storage management: New features overview. White Paper, 2007.

[Ora10a] Oracle Corporation. Oracle database concepts 11g release 2 (11.2), 2010. 14 Memory Architecture (Part Number E10713-05).

[Ora10b] Oracle Corporation. Oracle Database SQL Language Reference 11g release 2 (11.2), 2010. 3 Basic Elements of Oracle SQL (Part Number E26088-03).

[Ora10c] Oracle Corporation. Oracle Database SQL Language Reference 11g release 2 (11.2), 2010. 19 Concepts for Database Developers (Part Number E25789-01).

[Ora10d] Oracle Corporation. Oracle performance tuning guide 11g release 2 (11.2), 2010. 12 Using EXPLAIN PLAN (Part Number E10821-05).

[ÖV11] M. Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*. Springer, Heidelberg, Germany, 3rd edition, 2011.

[PG63] Noah S. Prywes and Harry J. Gray. The organization of a multilist-type associative memory. *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics (now IEEE Transaction on Communications)) (TAIEE (TCOM))*, 82(4):488–492, September 1963. IFIP article from 1962 - no electronic version available.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). *ACM SIGMOD Record (SIGMOD Rec.)*, 17(3):109–116, June 1988.

[Pis95] David Pisinger. *Algorithms for Knapsack Problems.* PhD thesis, University of Copenhagen, Copenhagen, Denmark, 1995. Adviser: Jakub Krarup.

[Pla09] Hasso Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1–2, New York, USA, 2009. Association for Computing Machinery.

[Pla11] Hasso Plattner. SanssouciDB: An in-memory database for processing enterprise workloads. In *Proceedings of the GI-conference on Database Systems in Business, Technology and Web (BTW)*, pages 2–21, Bonn, Germany, 2011. German Informatics Society.

[PMS99] Esther Pacitti, Pascale Minet, and Eric Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 126–137, San Francisco, USA, 1999. Morgan Kaufmann.

[Pol08] Vahe V. Poladyan. *Tailoring configuration to user's tasks under uncertainty.* PhD thesis, Carnegie Mellon University, Ann Arbor, USA, 2008. Adviser: Shaw, Mary; Garlan David.

[Raa93] Kimmo E. E. Raatikainen. Cluster analysis and workload classification. *SIGMETRICS Performance Evaluation Review (SIGMETRICS PER)*, 20(4):24–30, 1993.

[RAB+13] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. DB2 with BLU acceleration: So much more than just a column store. *The Proceedings of the VLDB Endowment (PVLDB)*, 6(11):1080–1091, August 2013.

[RBSS02] Uwe Röhm, Klemens Böhm, Hans-Jörg Schek, and Heiko Schuldt. FAS: A freshness-sensitive coordination middleware for a cluster of OLAP components. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 754–765. VLDB Endowment, 2002.

[Red11] T. Agami Reddy. *Applied data analysis and modeling for energy engineers and scientists.* Springer, Heidelberg, Germany, 2011.

[Riv76] Ronald L. Rivest. Partial-match retrieval algorithms. *Society for Industrial and Applied Mathematics Journal on Computing (SIAM J. Comput.)*, 5(1):19–50, March 1976.

[RKB87] Mark A. Roth, Henry F. Korth, and Don S. Batory. SQL/NF: A query language for ¬1NF relational databases. *Information Systems (Inf. Syst.)*, 12(1):99–114, January 1987.

[RS91] Steve Rozen and Dennis Shasha. A framework for automating physical database design. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 401–411, San Francisco, USA, 1991. Morgan Kaufmann.

[RWM02] R-Databases Special Interest Group, Hadley Wickham, and Kirill Müller. A common database interface (DBI). White Paper, 2002. Version 0.2-6, Updated: 21 April 2013.

[SAB+05] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel R. Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-Store: A column-oriented DBMS. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 553–564, New York, USA, 2005. Association for Computing Machinery.

[SAC+79] Pat Griffiths Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 23–34, New York, USA, 1979. Association for Computing Machinery.

[SB08] Ricardo Jorge Santos and Jorge Bernardino. Real-time data warehouse loading methodology. In *Proceedings of the International Database Engineering & Applications Symposium (IDEAS)*, pages 49–58, New York, USA, 2008. Association for Computing Machinery.

[SBKZ08] Jan Schaffner, Anja Bog, Jens Krüger, and Alexander Zeier. A hybrid row-column OLTP database architecture for operational reporting. In *Informal Proceedings of the International VLDB Workshop on Business Intelligence for the Real-Time Enterprise (BIRTE)*. VLDB website, 2008. `http://www.vldb.org/conf/2008/workshops/WProc_BIRTE/p7-schaffner.pdf`.

[Sc05] Michael Stonebraker and Ugur Çetintemel. One size fits all: An idea whose time has come and gone. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 2–11, Washington D.C., USA, 2005. IEEE Computer Society.

[Sch75] Mario Schkolnick. The optimal selection of secondary indices for files. *Information Systems (Inf. Syst.)*, 1(4):141–146, 1975.

[Sch91] Christoph Schneeweiß. *Planung 1, Systemanalytische und entscheidungstheoretische Grundlagen.* Springer, Heidelberg, Germany, 1st edition, 1991.

[Sch03] Alexander Schrijver. *Combinatorial optimization - Polyhedra and efficiency.* Springer, Heidelberg, Germany, 2003.

[SDJL96] Divesh Srivastava, Shaul Dar, Hosagrahar V. Jagadish, and Alon Y. Levy. Answering queries with aggregation using views. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 318–329, San Francisco, USA, 1996. Morgan Kaufmann.

[Sel88] Timos K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, 1988.

[SFA+16] Reza Sherkat, Colin Florendo, Mihnea Andrei, Anil K. Goel, Anisoara Nica, Peter Bumbulis, Ivan Schreter, Günter Radestock, Christian Bensberg, Daniel Booss, and Heiko Gerwens. Page as you go: piecewise columnar access in SAP HANA. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1295–1306, New York, USA, 2016. Association for Computing Machinery.

[SGS03] Kai-Uwe Sattler, Ingolf Geist, and Eike Schallehn. QUIET: Continuous query-driven index tuning. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1129–1132, San Francisco, USA, 2003. Morgan Kaufmann.

[SGS+13] Martin Schäler, Alexander Grebhahn, Reimar Schröter, Sandro Schulze, Veit Köppen, and Gunter Saake. QuEval: Beyond high-dimensional indexing à la carte. *The Proceedings of the VLDB Endowment (PVLDB)*, 6(14):1654–1665, September 2013.

[SHS05] Gunter Saake, Andreas Heuer, and Kai-Uwe Sattler. *Datenbanken: Implementierungstechniken.* mitp Verlag, Bonn, Germany, 2nd edition, 2005.

[SHWK76] Michael Stonebraker, Gerald Held, Eugene Wong, and Peter Kreps. The design and implementation of INGRES. *ACM Transactions on Database Systems (TODS)*, 1(3):189–222, September 1976.

[SSG04] Kai-Uwe Sattler, Eike Schallehn, and Ingolf Geist. Autonomous query-driven index tuning. In *Proceedings of the International Database Engineering & Applications Symposium (IDEAS)*, pages 439–448, Washington D.C., USA, 2004. IEEE Computer Society.

[SSG05] Kai-Uwe Sattler, Eike Schallehn, and Ingolf Geist. Towards indexing schemes for self-tuning DBMS. In *Proceedings of the International Conference on Data Engineering Workshops (ICDE Workshops)*, pages 1216–1223, Washington D.C., USA, 2005. IEEE Computer Society.

[SSV96] Peter Scheuermann, Junho Shim, and Radek Vingralek. WATCHMAN: A data warehouse intelligent cache manager. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 51–62, San Francisco, USA, 1996. Morgan Kaufmann.

[Sus63] Edward H. Sussenguth, Jr. Use of tree structures for processing files. *Communications of the ACM (CACM)*, 6(5):272–279, May 1963.

[SWES08] Dominik Ślęzak, Jakub Wróblewski, Victoria Eastwood, and Piotr Synak. Brighthouse: An analytic data warehouse for ad-hoc queries. *The Proceedings of the VLDB Endowment (PVLDB)*, 1(2):1337–1345, 2008.

[Syb10] Sybase Inc. *Sybase IQ 15.2 - Reference: Building, blocks, tables, and procedures*. Sybase Inc., Dublin, Ireland, 2010.

[SZ79] Prabhakant Sinha and Andris A. Zoltners. The multiple-choice knapsack problem. *Journal of the Operations Research (Operations Research)*, 27(3):503–515, 1979.

[Tan08] Frederick Tang. Size of NUMBER. `http://fredericktang.wordpress.com/2008/07/10/size-of-number/`, 2008. Visited May 2013.

[THC79] M. J. Turner, R. Hammond, and P. Cotton. A DBMS for large statistical databases. In *Very Large Data Bases, 1979. Fifth International Conference on (VLDB)*, pages 319–327, Washington D.C., USA, October 1979. IEEE Computer Society.

[Tra08] Transaction Processing Performance Council. TPC BENCHMARK^{TM} H. White Paper, 2008. Decision Support Standard Specification, Revision 2.8.0.

[Tra10] Transaction Processing Performance Council. TPC BENCHMARK^{TM} H. White Paper, 2010. Decision Support Standard Specification, Revision 2.11.0.

[Tur88] Carolyn Turbyfill. Disk performance and access patterns for mixed database workloads. *IEEE Data Engineering Bulletin (Data Eng. Bull.)*, 11(1):48–54, 1988.

[Val87] Patrick Valduriez. Join indices. *ACM Transactions on Database Systems (TODS)*, 12(2):218–246, June 1987.

[Van01] John Vandermay. Considerations for building a real-time data warehouse. White Paper, 2001. `http://www.grcdi.nl/considerations.pdf` (viewed October 2016).

[VMRC04] Alejandro A. Vaisman, Alberto O. Mendelzon, Walter Ruaro, and Sergio G. Cymerman. Supporting dimension updates in an OLAP server. *Information Systems (Inf. Syst.)*, 29(2):165–185, 2004.

[vNM53] John von Neumann and Oskar Morgenstern. *Theory of games and economic behavior*. Princeton University Press, Princeton, USA, 3rd edition, 1953.

[vWE86] Detlef von Winterfeldt and Ward Edwards. *Decision analysis and behavior research*. Cambridge University Press, New York, USA, 1986.

[VZZ+00] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Allan Skelley. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 101–110, Washington D.C., USA, 2000. IEEE Computer Society.

[WFW75] Gio Wiederhold, James F. Fries, and Stephen Weyl. Structured organization of clinical data bases. In *American Federation of Information Processing Societies: Proceedings of the National Computer Conference (AFIPS)*, pages 479–485, Montvale, USA, 1975. American Federation of Information Processing Societies Press.

[WGH+14] Johannes Wust, Martin Grund, Kai Hoewelmeyer, David Schwalb, and Hasso Plattner. Concurrent execution of mixed enterprise workloads on in-memory databases. In *Proceedings of the International Conference on Database Systems for Advanced Applications, Part I (DASFAA)*, volume 8421 of *Lecture Notes in Computer Science*, pages 126–140, Heidelberg, Germany, 2014. Springer.

[WGP13] Johannes Wust, Martin Grund, and Hasso Plattner. TAMEX: A task-based query execution framework for mixed enterprise workloads on in-memory databases. In *Informatics 2013: Informatics adapted to Men, Organization, and Environment (Informatics)*, volume 220 of *GI-Edition: Lecture Notes in Informatics*, pages 487–501, Bonn, Germany, 2013. German Informatics Society.

[WHM+93] Gerhard Weikum, Christof Hasse, Axel Mönkeberg, Michael Rys, and Peter Zabback. The COMFORT project. In *Proceedings of the International Conference on Parallel and Distributed Information Systems – Issues, Architectures, and Algorithms (PDIS)*, pages 158–161, Washington D.C., USA, 1993. IEEE Computer Society.

[WHMZ94] Gerhard Weikum, Christof Hasse, Alex Mönkeberg, and Peter Zabback. The COMFORT automatic tuning project, invited project review. *Information Systems (Inf. Syst.)*, 19(5):381–432, 1994.

[WKKS99] Gerhard Weikum, Arnd C. König, Achim Kraiss, and Markus Sinnwell. Towards self-tuning memory management for data servers. *IEEE Data Engineering Bulletin (Data Eng. Bull.)*, 22:3–11, 1999.

[WKLS12] Thorsten Winsemann, Veit Köppen, Andreas Lübcke, and Gunter Saake. A layered architecture approach for large-scale data warehouse systems. In *Proceedings of the International United Information Systems Conference (UNISCON)*, number 137 in Lecture Notes in Business Information Processing, pages 199–201, Heidelberg, Germany, 2012. Springer.

[WMHZ02] Gerhard Weikum, Axel Mönkeberg, Christof Hasse, and Peter Zabback. Self-tuning database technology and information services: From wishful thinking to viable engineering. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 20–31, San Francisco, USA, 2002. Morgan Kaufmann.

[Wor12] Jeff Word. *SAP HANA essentials*. Epistemy Press, Frisco, USA, 1st edition, 2012.

[WY76] Eugene Wong and Karel Youssefi. Decomposition – A strategy for query processing. *ACM Transactions on Database Systems (TODS)*, 1(3):223–241, 1976.

[ZAL08] Youchan Zhu, Lei An, and Shuangxi Liu. Data updating and query in real-time data warehouse system. In *Proceedings of the International Conference on Computer Science and Software Engineering (CSSE)*, pages 1295–1297, Washington D.C., USA, 2008. IEEE Computer Society.

[ZBNH05] Marcin Zukowski, Peter A. Boncz, Niels Nes, and S. Heman. MonetDB/X100 - A DBMS in the CPU cache. *IEEE Data Engineering Bulletin (Data Eng. Bull.)*, 28(2):17–22, 2005.

[ZCL+00] Markos Zaharioudakis, Roberta Cochrane, George Lapis, Hamid Pirahesh, and Monica Urata. Answering complex SQL queries using automatic summary tables. *ACM SIGMOD Record (SIGMOD Rec.)*, 29(2):105–116, 2000.

[ZNB08] Marcin Zukowski, Niels Nes, and Peter A. Boncz. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, pages 47–54, New York, USA, 2008. Association for Computing Machinery.

[ZRL+04] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam J. Storm, Christian Garcia-Arellano, and Scott Fadden. DB2 Design Advisor: Integrated automatic physical database design. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1087–1097, San Francisco, USA, 2004. Morgan Kaufmann.

[ZZL+04] Daniel C. Zilio, Calisto Zuzarte, Sam Lightstone, Wenbin Ma, Guy M. Lohman, Roberta Cochrane, Hamid Pirahesh, Latha S. Colby, Jarek Gryz, Eric Alton, Dongming Liang, and Gary Valentin. Recommending materialized views and indexes with IBM DB2 design advisor. In *Proceedings of the International Conference on Autonomic Computing (ICAC)*, pages 180–188, Washington D.C., USA, 2004. IEEE Computer Society.