Ph.D. Thesis

TECHNISCHE UNIVERSITÄT DRESDEN

Flexibility in Data Management

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technische Universität Dresden
Fakultät Informatik

eingereicht von
Dipl.-Inf. Hannes Voigt
geboren am 14. Dezember 1980 in Leipzig

Gutachter: **Prof. Dr.-Ing. Wolfgang Lehner**
Technische Universität Dresden
Fakultät Informatik, Institut für Systemarchitektur
Lehrstuhl für Datenbanken
01062 Dresden, Germany

Dr.-Ing. Christof Bornhövd
Chief Development Architect
HANA Product & Development
SAP Labs, LLC
3420 Hillview Avenue, Palo Alto, CA 94304, USA

Tag der Verteidigung: 3. März 2014

Dresden im März 2014

Abstract

With the ongoing expansion of information technology, new fields of application requiring data management emerge virtually every day. In our knowledge culture increasing amounts of data and work force organized in more creativity-oriented ways also radically change traditional fields of application and question established assumptions about data management. For instance, investigative analytics and agile software development move towards a very agile and flexible handling of data. As the primary facilitators of data management, database systems have to reflect and support these developments. However, traditional database management technology, in particular relational database systems, is built on assumptions of relatively stable application domains. The need to model all data up front in a prescriptive database schema earned relational database management systems the reputation among developers of being inflexible, dated, and cumbersome to work with. Nevertheless, relational systems still dominate the database market. They are a proven, standardized, and interoperable technology, well-known in IT departments with a work force of experienced and trained developers and administrators.

This thesis aims at resolving the growing contradiction between the popularity and omnipresence of relational systems in companies and their increasingly bad reputation among developers. It adapts relational database technology towards more agility and flexibility. We envision a descriptive schema-comes-second relational database system, which is entity-oriented instead of schema-oriented; descriptive rather than prescriptive. The thesis provides four main contributions: (1) a flexible relational data model, which frees relational data management from having a prescriptive schema; (2) autonomous physical entity domains, which partition self-descriptive data according to their schema properties for better query performance; (3) a freely adjustable storage engine, which allows adapting the physical data layout used to properties of the data and of the workload; and (4) a self-managed indexing infrastructure, which autonomously collects and adapts index information under the presence of dynamic workloads and evolving schemas. The flexible relational data model is the thesis' central contribution. It describes the functional appearance of the descriptive schema-comes-second relational database system. The other three contributions improve components in the architecture of database management systems to increase the query performance and the manageability of descriptive schema-comes-second relational database systems. We are confident that these four contributions can help paving the way to a more flexible future for relational database management technology.

Acknowledgements

This thesis would not have been possible without the support, advice, and encouragement of a large number of people. First, I would like to thank Wolfgang Lehner, my supervisor, who gave me the opportunity to pursue a doctorate and supported me along the way with advice, an outstanding group of colleagues and a window seat in the office. Wolfgang leads by example, by living and showing his standards as a matter of course, as the most normal and natural thing to do. This always gave me the confidence and faith in being able to live up to these standards and accomplish the goal of a doctorate, which seems so large and far away in the very beginning. I am very grateful for the patience, confidence, and advice Wolfgang always had, especially during the time it took me to find a topic and get productive. Every day I am in the office (or home office) I very much appreciate the open, liberal and at the same time demanding and fostering working atmosphere Wolfgang has established in his research group based on trust, cooperation, friendship, and standards.

Equally, I would like to thank Christof Bornhövd. Christof has accompanied me in my endeavor ever since April 2009. Right from our first meeting, Christof has fed my interest in schema-comes-second data management. He is co-refereeing this thesis and provided elaborate and detailed comments whilst it has been in the making. From July 2010 to June 2011 during my internship year at SAP in Palo Alto, Christof was a welcoming and generous host. His hospitality played a big part in making these twelve months abroad an experience that will last me lifetime.

I am very thankful to Anthony Creaton for taking the pain in proofreading the thesis, as well as to Katrin Braunschweig, Ulrike Fischer, Marcus Paradise, Michael Rudolf, Ulrike Schöbel, and Maik Thiele – their valuable comments on various parts of the thesis were of great help. I would like to thank Johannes Fenzel, Robert Kubis, Thomas Kissinger, Kai Herrmann, Alfred Hanisch, Tobias Jäkel, Linda Jantschke, Henry Mühle, Marcus Paradise, and Michael Rudolf who worked with me on topics directly part of, or closely related to, the thesis; Dirk, Frank, Thomas, and Uli for being great roommates at the office; Anja, Benjamin, Bernd, Bernhard, Claudio, David, Dirk, Elena, Eric, Frank, Gunnar, Henry, Ines, Ismail, Jens, Julian, Kai, Kasun, Katrin, Lars, Maik, Marcus, Markus, Martin, Matthias, Michael, Peter, Philipp G., Philipp R., Rainer, Rihan, Robert, Simone, Steffen, Tobias, Tim, Till, Thomas L., Thomas K., Tomas, Uli, and Ulrike for being great colleagues and friends.

None of my doings would have been possible, though, without the constant and continuing support of my family and friends. I am deeply grateful to my grandparents and my parents, who have always been on my side and supported me wherever they could, never losing confidence in me. I thank all my friends for the great times we had together here in Dresden, in Leipzig, in Berlin, in California, and wherever we

traveled. Finally but most importantly, I thank my love Madlen. Although she is always, in her own lovely way, the first to foster and emphasize the non-science part of my life by drawing me to and challenging me on her fields of interest, she supported me and accepted it when I had to work during what little time we have together in any case.

Hannes Voigt
Dresden, March 5, 2014

Contents

1 Introduction

Relational databases are the foundation of western civilization.
Bruce Lindsay [Winslett, 2005]

Today's databases live in diverse and changing ecosystems. The functional and non-functional requirements we design a database for become increasingly unstable. End user empowerment [Nagarajan, 2011], agile analytics [Cohen et al., 2009], agile software development methods [Beck et al., 2001] and data integration [Franklin et al., 2005, Sarma et al., 2008] are particular drivers of this development. Yet, traditional relational data management builds on the assumption of rather stable requirements. As a consequence, many developers perceive relational data management technologies as cumbersome, inflexible and dated [Kiely and Fitzgerald, 2005]. Instead of the strict, prescriptive schema-comes-first thinking in traditional data management, a flexible and descriptive schema-comes-second take on data management is more suitable for many of today's application areas. The current momentum in the NoSQL development gives evidence.

Still, relational database management systems are the most common in the IT departments around the world. Relational systems account for 90 % of the information systems in Fortune 100 companies [Brodie and Liu, 2010]. There are a number of good reasons for many businesses to stick with relational systems [Leavitt, 2010, Stonebraker, 2011, Brodie and Liu, 2010, Mohan, 2013]. Relational database management systems bring 30 years of proven database technology. They are powerful and easily available. Their concepts are widely and well known, with hundreds of thousands of experienced and trained developers and administrators around the world. Relational database management systems build on established industry standards and are interoperable and a long known player in complex IT infrastructures. In addition, there is a wide set of mature tools and utilities around relational databases helping with design, development, administration, and maintenance.

The wide gap between the still existing popularity of relational systems and their misfit with current realities in application domains and software development is obvious. This thesis is an effort to close this gap. Its aim is to adapt relational database technology to schema-comes-second data management – to remove the cumbersomeness and inflexibility of relational database technologies while retaining its strengths, power, standards, maturity, and interoperability.

In the remainder of this introduction, we briefly outline the basics of traditional schema-comes-first data management in Section 1.1, where relational database management systems originate from. In Section 1.2 we contrast this with an elaborate description of schema-comes-second data management to show where the deficits of

traditional relational database systems are. Section 1.3 closes the introduction with an overview of the thesis and its contributions.

1.1 Traditional Schema-Comes-First Data Management

Traditional database management systems, typically relational, are widely available today. They can be purchased as products or licenses from bigger and smaller vendors, such as Oracle, IBM, Microsoft, Sybase, and SAP. Alongside commercial products there exist database management systems, such as MySQL, PostgreSQL, Derby, and H2, which are available for free. Database management systems implement data management functionality in a general, application-independent way. A database management system is not immediately ready to provide functionality for a particular database. Rather, they have to be set up according to the requirements the database has to fulfill.

Database systems are complex and have to meet many functional and non-functional requirements. Every application scenario has its own very specific requirements. Functional requirements include aspects of the database schema, the workload, and access rights. In contrast, non-functional requirements involve rather technical parameters such as expected data size, tolerated latency, throughput, and concurrency. To properly set up a database management system, all requirements have to be known. Requirement analysis is complicated by the fact that requirements of multiple applications can differ or can even be contradictory.

In the 1970s, database research and industry established a database design process that is still used today. That process guides developers in systematically collecting requirements and helps in translating them step-by-step into a complete database configuration. The process consists of four phases: analysis, conceptual design, logical design, and physical design [cf. Liu and Özsu, 2009, Database Design]. During analysis, developers collect and structure application-specific requirements, typically extracted from discourse descriptions. In the conceptual design phase, the different requirements are consolidated into a single conceptual design. This is based on a conceptual model language, such as Entity-Relationship [Chen, 1975] or UML [OMG, 2009], and is independent of any data model implemented in database management systems. The logical database design transforms the conceptual design to an implemented data model of choice, such as a relational [Codd, 1970], object-orientated, or graph-based [Rodriguez and Neubauer, 2010] data model. The result of this transformation is a logical design. Developers create the logical design through well-defined transformation rules from the conceptual design [Teorey et al., 1986]. To compensate for mistakes made in the conceptual design, the logical design phase can involve normalization procedures [Bernstein, 1976]. All functional requirements have to be met by the logical design. In the final physical design phase, the logical design is implemented in a specific database management system on specific hardware. Here, the developers decide, among many other parameters, on the configuration of access paths, such as indexes, partitions, and materialized views. During the physical design,

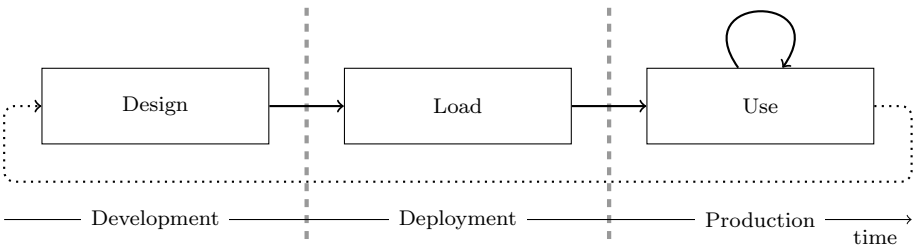


Figure 1.1: Traditional database usage pattern

the developer's decisions are based on the expected data and workload, i.e. the most important queries and updates. The result is the physical design of the system; that also has to meet all non-functional requirements. The whole design process is not strictly sequential, but rather iterative. At any time, developers may loop back to an earlier phase to revise the design. The goal is a complete database design that meets all functional and non-functional requirements.

Figure 1.1 shows the traditional usage pattern of a database. Designing the database and loading data typically happens at development and deployment, before the database is used productively. The whole design process typically takes multiple days or weeks depending on the complexity of the database-to-be. If a stable design is found that can be productively used for a long time, the effort pays off. Traditional database applications, such as banking, accounting, resource management, and business reporting, allow a stable database design. However, once in a while even stable database designs have to be adapted or extended, for example as the applications built on top evolve. At this this point, the usage pattern is basically repeated, including the laborious design process. After a redesign, loading data includes transfering the data from the old design to the new design, which is typically a delicate process requiring a lot of attention and care to avoid data loss. Again, if the new design is stable for a reasonably long time, the efforts spent on the redesign can pay off. With the traditional usage pattern, stable database design as well as stable requirements are crucial for the economic efficiency of a database.

1.2 Modern Schema-Comes-Second Data Management

Since the beginning of the 21st century we have perceived a quickly progressing digitalization towards a complete digital transformation of society, business, science, and culture. Electronic information technology gradually pervades every part of our civilization. In 2007, 94 % of the world's technological memory and 99 % of the world's telecommunication was digital [Hilbert and López, 2011]. Text, audio, and film – each of the three major types of media – transformed to a digital electronic representation. We capture, edit, distribute, consume, and archive media content predominantly by means of electronic technology. Likewise, all kinds of physical

parameters (e.g., position, location, distance, speed, pressure, weight, temperature, current) can be measured digitally. With electronic technology, we observe, measure, and analyze real-world events and control activities and processes. Databases are the prime means of handling larger amounts of electronic information. With the ongoing proliferation of information technology, databases inevitably appear everywhere where we start accumulating electronic information for archiving, manipulation, controlling, intelligence, and analysis.

The success of electronic technology feeds its proliferation as well as its development Kurzweil [2001]. Information technology is known for its exponential growth rates. The main facilitators of these growth rates are the steady advancements in semiconductor manufacturing, famously summarized in Moore's law [Moore, 1965]. The net effects are visible on a global scale. Our worldwide capacity to store information per capita grew on average by 25 % per year between 1986 and 2007. In the same time, our general purpose computing capacity per capita grew on average by 58 % per year [Hilbert and López, 2011]. While advancements in manufacturing allow increasing production of storage and computing capacities each year, they also lower the price per unit. The ubiquitous availability of cheap storage, computing, and communication capacities greatly stimulates creativity in putting available information to use and constantly propels new applications and usage patterns.

Our world is driven by ideas. Algorithmic routine tasks are either automated or outsourced to developing countries. First and foremost, the Western job markets prosper around complex, interactive, creative, and heuristic work [Pink, 2006, 2009]. In 2005, McKinsey categorized 70 % of the jobs created in the United States between 1998 and 2004 as predominately characterized by complex interactions [Johnson et al., 2005]. Creativity, ideas, and inventions are per se unpredictable [Taleb, 2010]. Predicting the ideas of tomorrow requires having them today. Learning follows an unscripted process. Understanding forms and evolves on the fly and may never reach a fixpoint. In consequence, what we do with databases – how we organize, retrieve, and analyze information – is a constantly evolving and inherently unpredictable field.

Today's database management systems are used in diverse and changing fields of application. With the persistent acceleration of society [Gleick, 1999] these trends will more likely intensify than regress. Soon we will see databases in unexpected areas fueling applications we have not thought of before. The requirements for which we design a database become increasingly unstable. Application domains with unstable requirements demand developers to constantly remodel databases, and in particular database schemas. While such agile application domains used to be the exception, they are common today. Monash recently compiled a not-meant-to-be-complete list of common reasons causing database schemas to change [Monash, 2013a]; among those are every day events such as product and service changes, organizational changes, mergers and acquisitions. 30 % of all information systems in Fortune 100 companies are modified significantly every year [Brodie and Liu, 2010]. Given the laborious database design process, it is more than clear that many developers perceive relational data management technologies and database design methods as cumbersome, inflexible and dated [Kiely and Fitzgerald, 2005]. The prescriptiveness of the traditional database

schema considerably limits the agility and flexibility of a database system with respect to changes in its application domain. In the following, we will describe two areas of application for database systems that are characterized by very agile database usage.

1.2.1 Investigative Analytics

Knowledge is the central element of today's economic growth and revenue generation [Powell and Snellman, 2004]. With the powerful computing and storage technology that is available as a commodity today, data analysis has become an essential means to gain new knowledge. It requires asking the right questions, and having the right data available to answer them. Traditionally, with the goal of helping business controlling and providing decision support, these have been questions for weekly and monthly reports on business data available in a company's ERP and CRM systems. The established data analytics technology has been developed for such weekly and monthly reports. Typically, a company integrates all necessary data sources into a comprehensive and consolidated data warehouse and runs batches of queries to generate the reports [Lehner, 2002]. For controlling and decision support, data warehouses have been an effective and successful solution, accounting for a considerable part of the success of traditional data management technology in general.

However, the world of analytics has changed. Today, we have more data available than ever before. In 2008, an IDC report [Gantz et al., 2008] estimated nearly 1800 exabytes of digital information to be produced in 2011 alone. This is ten times the amount produced in 2006, resulting in a compound annual growth rate of almost 60 %. About half of the produced information gets stored and is of potential interest for analysis and knowledge discovery. A 2012 survey among companies running Oracle database systems found that "42 % can be considered large data shops, supporting more than 50 TB" [McKendrick, 2012]. Especially sentiments-carrying data is of analytical interest, most prominently marketing campaign data, social data, internet log data, and data derived from other analyses [Stodder, 2012]. Often such data comes from partners and is not produced in-house. Tapping all these sources is crucial on the way to new valuable knowledge. However, all of these kinds of data are typically very dynamic and prone to schema changes [Monash, 2013a].

Today, we are also better and more creative in analyzing the data. Data analysis has become a common practice to gain knowledge throughout all business departments; as well as in science [Hey et al., 2009] and even art [Viégas and Wattenberg, 2007]. With the spread of data analysis, the methods used have greatly advanced. While traditional analytics involved mainly monitoring and reporting, investigative analytics is "seeking (previously unknown) patterns in data" [Monash, 2011]. The demand for new knowledge is not satisfied by setting up monthly or weekly reports. The work of knowledge workers, analysts, and data scientists becomes increasingly important. In a recent survey, 61 % of the respondents said the data scientist/analyst job role is essential to their organization [McKendrick, 2013]. Analysts and data scientists use sophisticated algorithms and follow highly iterative workflows, where every answer sparks new questions [Endeca, 2008] or as Monash [2013b] phrases it: "People don't

want to submit requests for reports or statistical analyses; they want to get answers as soon as the questions come to mind." Similar workflows have also been framed with the notions of dataspaces and pay-as-you-go data integration [Franklin et al., 2005, Sarma et al., 2008].

Having been the initial trailblazer for analytics, traditional data management technology becomes more and more the impediment to the quest for up-to-the-minute knowledge. Where our understanding of data constantly evolves, schema changes are inevitable. The properties of database management systems for investigative analytics were famously described by Cohen et al. as MAD skills [Cohen et al., 2009]. Accordingly, a database management system has to be *magnetic* in "attracting all the data sources that crop up [...] regardless of data quality niceties", *agile* as a "database whose physical and logical contents can be in continuous rapid evolution", and *deep* by serving "both as a deep data repository and as a sophisticated algorithmic runtime engine". Traditional relational database management systems fundamentally lack these properties required in the world of investigative analytics. In particular, they lack the flexibility for handling dynamically evolving schemas.

1.2.2 Agile Software Development

Before the 1960s software development did not follow any particular pattern or formalized method. Solely focussed on and trained in the technical side of software programming, developers followed an individual approach and rarely understood the business context of an application or user needs. As this hampered the overall progress in software development, organizations started to establish more disciplined approaches to software development. One of the first models used to structure the development of information systems is the system development life cycle, also known as the waterfall model. It identified sequential stages of planning, analysis, design, development, testing, and so on. The purely sequential nature of the waterfall model still led many projects to fail to satisfy the users. Users were typically rarely involved, and when they were, then mostly too late. Incremental development methods had broader success. With it, the whole concept of a development method evolved to a "recommended collection of phases, procedures, rules, techniques, tools, documentation, management, and training used to develop a system" [Avison and Fitzgerald, 2003].

Since the late 1990s, however, incremental approaches specifically and methods in general started being perceived as rigid, inflexible, ritualistic, and creativity inhibiting [Wastell, 1996]. By now, software development has widely changed to agile and lean approaches such as extreme programming (XP) [Beck, 1999], Scrum [Takeuchi and Nonaka, 1986, Schwaber, 1997], Crystal Clear [Cockburn, 2004], and Kanban [Ohno, 1988, Anderson, 2003, 2010]. For an overview see Abrahamsson et al. [2003] and Cohen et al. [2004]. A 2012 survey among 4048 individuals from the software development community found that 84 % of the respondents' organizations were practicing agile development, a growth of 80 % from 2011 [VersionOne, 2012]. In the same study Scrum appeared to be by far the most popular method; 72 % of the respondents' organizations were using Scrum or Scrum variants.

Although the agile methods differ in the particular organization and controlling of the software development process, they all share commonalities. The manifesto for agile software development [Beck et al., 2001] states twelve principles subsuming the idea of agile methods. Among those: "welcome changing requirements", "deliver working software frequently", "working software is the primary measure", and "best architectures, requirements, and designs emerge from self-organizing teams". Instead of involving developers in rigid processes and extensive planning, agile software development centers the creativity and excellence of people to handle the unpredictable dynamic world of software development. Agile methods are characterized by short development cycles, each with the goal of a shippable product. In Scrum, for instance, a sprint typically lasts between one week and one month. Teams are small, self-organizing groups of developers, usually not larger than ten individuals. Priorities can be readjusted on a daily basis.

The traditional relational database development procedure is obviously incompatible with the aim of agile software development. Schema-comes-first thinking prescribes the design of a comprehensive, elaborate schema. This sequential procedure originates from the 1970s and fits well with the waterfall model and incremental methods of software development, but is not able to keep up with the speed and flexibility of agile development. Unfortunately, the design of relational database management systems also originates from the 1970s and still carries the thinking of the sequential development procedure prevailing in those years. Relational database management systems' DNA is simply not agile.

1.2.3 Summary

Agile application domains demand a descriptive schema-comes-second way of thinking in data management. This implies a new agile database usage radically different from the traditional one. While the traditional Design-Load-Use pattern builds on consecutive steps, they are rather concurrent and iterative in the agile database usage. Data is loaded from other sources and is directly used. When needed, data can be redesigned or more data can be loaded from other sources. Where the traditional pattern separates design time and production time, the agile usage pattern considers everything as being part of production time. The database schema is not developed separately; it evolves with the database usage. The schema is explicitly user-driven. While working with the data, the user changes the database schema whenever necessary so that it follows the user's current understanding and needs. Traditional schema adaptation mechanisms such as views do not represent adequate solutions since they offer only additional prescriptive schema where descriptive schema is required. In agile application domains, the schema is the user's means to describe the structure of data, rather than the database's means to prescribe the structure of data. This may lead to ambiguous and inconsistent data. However instead of preventing ambiguity and inconsistencies, however, the database management system should support the user in finding and fixing them.

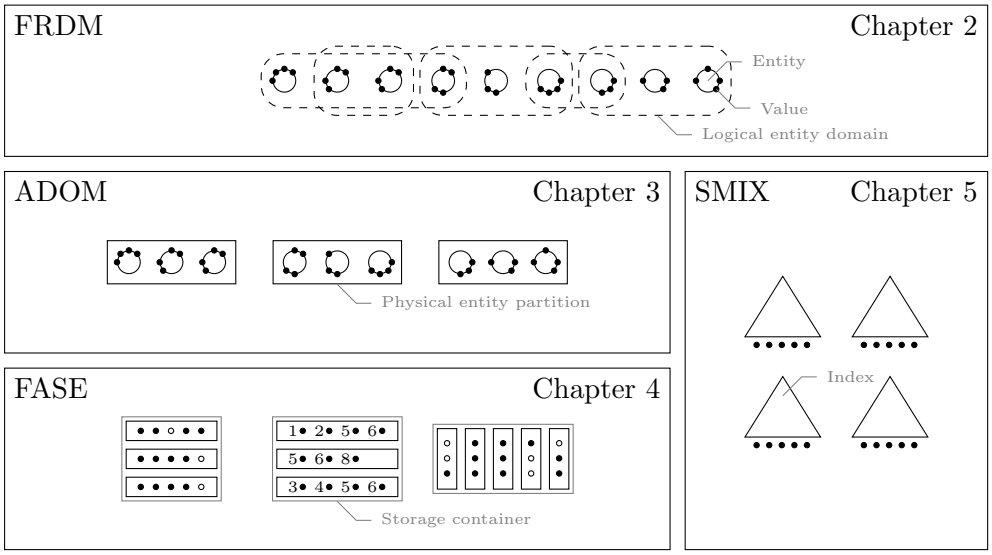


Figure 1.2: Contributions of the thesis.

1.3 Contributions

This thesis aims at increasing the flexibility of relational database systems to make them attractive for the agile world of today. Specifically, we make relational database technology ready for schema-comes-second data management. To achieve this goal, the thesis provides four contributions.

FRDM [ˈfriːdəm] a flexible relational data model.

ADOM [ˈædəm] autonomous physical entity domains.

FASE [ˈfeɪz] a freely adjustable storage engine.

SMIX [ˈsmɪks] a self-managed indexing infrastructure.

Figure 1.2 illustrates the four contributions of the thesis. The flexible relational data model FRDM is the thesis' central contribution towards schema-comes-second data management. FRDM is descriptive rather than prescriptive and allows a user-driven and freely evolving schema. FRDM gains its flexibility from omitting the implicit constraints that define the prescriptive nature of the relational data model. Among other flexibilities, FRDM allows tuples to have arbitrary attributes and be part of any number of tables (entity domains in the FRDM terminology). It is possible to implement FRDM in existing relational database management systems by using existing components and technologies. However, accompanied by the thesis' other three contributions, it makes a more complete and better-to-handle system. We discuss FRDM in detail in Chapter 2. Parts of the material in Chapter 2 have been developed jointly with Wolfgang Lehner.

In FRDM, an entity domain is a purely logical concept, which has a crucial effect on system architecture. In contrast to traditional relational tables, logical entity domains cannot double as physical entity domains for the storage system since entities may appear in multiple domains. The self-managed horizontal partitioning ADOM autonomously manages physical entity domains based on the self-description of the entities. It assigns each entity to a single physical domain, specifically to the domain where the entity fits best according to its schema. This physical entity separation allows the database system to increase its query efficiency. Instead of scanning the large pile of all entities in a database, queries can skip ADOM partitions that are irrelevant schema-wise. ADOM is directly integrated in the processing of the database system and works with low overhead. We discuss ADOM in detail in Chapter 3. Parts of the material in Chapter 3 have been developed jointly with Kai Herrmann and Wolfgang Lehner and have been published in an early version in Herrmann et al. [2014] and, in the version of the thesis, are accepted for publication to the 9th International Workshop on Self-Managing Database Systems 2014.

In FRDM, entities can have arbitrary attributes, which has an impact on the efficiency of the physical data layout. A couple of physical data layouts for self-descriptive, irregularly structured data have been proposed and discussed in the literature. However, the physical data layout most efficient for a given database depends on the actual irregularity and on the workload. FASE, therefore, provides a configurable physical data layout. This allows seting up different physical data layouts for databases in the same database management system, all using the same storage engine. FASE supports all common data layouts for regular data and irregular data. For each of those layouts, FASE exhibits the same performance trade-offs as one would expect from a native implementation. FASE significantly increases the physical data independence of database management systems. We discuss FASE in detail in Chapter 4. Parts of the material in Chapter 4 have been developed jointly with Alfred Hanisch and Wolfgang Lehner.

With the arbitrary attribute sets FRDM entities can have, the definition of useful indexes becomes a problem, too. Where the schema is not known in advance, indexes cannot be defined in advance. Where a self-descriptive schema constantly evolves, the index configuration has to evolve constantly, too. This is particularly important for secondary indexes, which represent additional access information and occupy additional resources. SMIX solves this problem. It provides an infrastructure of self-managed secondary indexes, where the DBA only allocates a fixed amount of resources for indexation and the database system takes care of the rest. SMIX incrementally builds up secondary index information during query processing for the data that is queried. To remain within the resource limits, SMIX also discards index information if it is not used anymore. SMIX can autonomously adapt to changing workloads as well as to changing data sets and changing schemas. We discuss SMIX in detail in Chapter 5. Parts of the material in Chapter 5 have been developed jointly with Thomas Kissinger, Tobias Jäkel and Wolfgang Lehner and have been published in Voigt et al. [2013], Kissinger et al. [2012], and Voigt et al. [2012].

Although ADOM, FASE, and SMIX are of particular help in a FRDM database system, all four contributions of this thesis are also valid independently of one another as they address very different aspects of database systems. We will, therefore, discuss each contribution, including related work and evaluation, separately in one of the following four chapters. The final chapter concludes the thesis and provides suggestions for future research.

2 FRDM – A Flexible Relational Data Model

> Any data-related business processes you have [...] should assume that your data models will be in perpetual, rapid flux.
>
> *Curt Monash [Monash, 2013a]*

The *Flexible Relational Data Model* (FRDM) is a descriptive schema-comes-second relational data model. It is entity-oriented instead of being schema-oriented; descriptive rather than prescriptive. The flexibility FRDM offers, is based on an assessment of the flexibility and inflexibility of the traditional relational data model. Specifically, FRDM allows:

- Self-descriptive entities: Entities can instantiate arbitrary sets of attributes regardless of the entity domain (a table in a traditional relational database). The attributes an entity instantiates can be changed any time.
- Multifaceted entities: Entities can belong to multiple entity domains. Entities can join domains or leave domains any time.
- Independent attributes: Attributes do not have to be assigned to a dedicated entity domain. Entities from different entity domains can instantiate the same attribute. Attributes can be added or dropped independently from entity domains.
- Independent technical types: Values from the same attribute can differ in their technical type (e.g., integer, float, char, etc.). The technical type of a value can be changed at runtime without affecting the attributes or other values of the attributes.

FRDM is relational in the sense that it builds on the operational power of the relational algebra. It is compatible and interoperable with relational data sets.

FRDM is accompanied by a flexible constraint framework FRDM-C that provides explicit restrictions on the flexibility of FRDM. The scope and the range of the restriction can be tailored to any requirements ranging from the constraint-free, descriptive nature of pure FRDM to the strictly prescriptive nature of the traditional relational data model. FRDM-C helps to introduce rigidity exactly when and at which parts of the data it is needed for the application at hand. FRDM-C constraints can vary in their effect from simply informing to strictly prohibiting, so that they are not only a tool to maintain data quality but also help with achieving data quality.

As a preliminary before detailing FRDM, we assess the relational data model regarding its flexibility and inflexibility in Section 2.1 and discuss other flexible data models in Section 2.2. Then, Section 2.3 and 2.4 present FRDM and FRDM-C in

detail, respectively. We show how relational data can be represented in FRDM and how FRDM can be restricted to the strictness of the relational data model with FRDM-C constraints in Section 2.5, to demonstrate the super-relational nature of FRDM. This is followed by considerations regarding the implementation of FRDM within the architecture of relational database management systems (Section 2.6). Finally, Section 2.7 concludes the chapter.

2.1 Assessment of the Relational Data Model

To many developers the relational data model appears cumbersome and inflexible. A clear understanding of the inflexibility of relational data model is a necessity to create a more flexible relational data model. In the first half of the following assessment we examine which particular inflexibilities the relational data model exhibits. Under closer observation, the relational data model also exhibits some valuable flexibilities, which are worthwhile to retain. We discuss these relational flexibilities in the second half of the assessment.

2.1.1 Relational Inflexibility

The perceived cumbersomeness and inflexibility of the relational model originates from implicit constraints. Unlike constraints explicitly specified in the schema, implicit constraints are not user-made; they represent inherent inflexibility of the data model. Implicit constraints forbid loading new inconsistent and ambiguous data into an existing database; implicit constraints hinder retrieving data in new compositions. All of these are good features if the corresponding behavior is desired. If not, however, the features turn into limitations. The essential problem of implicit constraints is that the user cannot switch them off; the user is committed to implicit constraints. Therefore, implicit constraints are incompatible with schema-comes-second data management. A flexible relational data model has to be free of implicit constraints; all constraints ought to be explicit and user-defined.

Implicit constraints of a data model limit the freedom of the user to structure the data. Nevertheless, the data model is the contract between user and database management system about which basic structure data will have. This elementary structure is the foundation for all operations and services a database management system offers. Consequently, a data model for structured data has to specify some elementary structure; without, it is a data model for unstructured data. The minimum degree of structure needed to refer to data as structured data is a controversial discrimination and depends on who is asked.

Nevertheless, to identify the implicit constraints of the relational data model, we define a baseline model for structured data. The baseline model comprises the model elements common to the major data models for structured data. That way, the baseline model ensures a minimmum degree of structure needed for structured data.

Baseline Model The common nucleus of general-purpose data models for structured data is the entity. An entity is a set of values that describe a simple real-world object or aspect. Each value belongs to a logical value domain, e.g. name, age, zip code, street no. Following the technical conditions in computer systems, the values are encoded according to technical types, such as integer, float, and string. Entity and value domain uniquely identify a value. For operations, entities need to be identifiable individually, e.g. the entity *Margaret Atwood*, and in logical groups, e.g. all *novelists*. Hence, entities have an identity for individual identification and belong to logical entity domains for group identification.

The relational model defines a significantly more rigid structure than the baseline model. In the baseline model entity, entity domain, value domain, and technical type are independent concepts used to group values along different aspects. In the relational model the four equivalent concepts tuple, relation, column, and technical type depend on each other. Relation as the central concept brings three functions together in one place. Relations are (1) entity containers for storage and processing, (2) entity domains, i.e. logical handles to address groups of entities, and (3) define entity schema constraints. The combination of these three functions into a single concept results in the subordination of the other data model concepts and, in consequence, results in the five implicit constraints that characterize the relational model.

Dependent values domains Relations define their columns. Columns cannot exist independently from relations. Two columns from different relations may have the same name, but they are not the same, they are considered two different value domains.

No multifaceted entities Relations define their tuples. Two tuples from different relations are considered different entities. The same tuple cannot be part of multiple relations. Only a copy of a tuple can be restored in another relation, still the copy and the original are treated as two individual tuples. Multi-faceted entities belonging to multiple entity domains are not possible.

No generalized entities Relations dictate which value domains their tuples have to instantiate. Tuples must fill all the value domains, and so are prevented from having a more general intension than the relation.

No specialized entities Besides generalization, the schema dictate of relations also prohibits specialization. Tuples are not allowed to have a specialized intension by filling additional value domains.

Dependent technical types Columns specify the technical type for all their values. Values in the same value domains cannot have different technical types in the relational model.

2.1.2 Relational Flexibility

While the relational model exhibits implicit constraints on domains and technical types, it is very flexible regarding entity constitution and entity relationships. In these two aspects, the relational model does not bear implicit constraints. This separates the relational model from other data models, such as the object-oriented data model, and is also one of the major reasons for its success in databases.

All data models define what constitutes a single entity by assigning an identity to each entity. This entity identity allows operations to refer to single entities and to establish a basis for entities referencing each other. There are different ways to establish an entity identity. The most common are:

Positional identity Entity identity depends on the entity's position in the sequence or in the hierarchy of a corpus of entities. Moving an entity to another position in the corpus changes its identity.

Object identity A fixed value domain serves as identity. The values of this domain are either generated by the system (e.g. object id) or provided by the user (e.g. URI). The identity can be explicitly visible to the user as a regular value or hidden by the system as in many object-oriented programming languages.

Value-based identity One or more value domains serves as identity. The set of value domains that constitute the identity depends on the context and is defined by the user.

Obviously, positional identity and object identity implicitly constrain the identity definition. What defines the identity of entities is fixed by the data model; it cannot be adapted in the context of a database or a query at runtime. The relational model uses value-based identity, which is free of implicit constraints. Here, the user defines what constitutes the identity of tuples, either with explicit primary constraint for base relations in the schema or by distinct or group operations for result relations in queries.

To represent relationships between entities, most data models allow entities to reference other entities. Again, there are different ways to achieve this. The most common referencing mechanisms are:

Nesting References are expressed with a hierarchical representation of entities, where entities are nested in other entities. By definition, nested entities are in a relationship. Cyclic references between multiple entities cannot be represented. Moving an entity to another position in the corpus changes the represented relationships.

Explicit association References are expressed with a dedicated association element defined by the data model. In the data, every reference is explicitly marked as such.

Value-based reference References are expressed by value equality. Two entities are in a relationship if they have the same value or similar values on certain value domains. The set of value domains that constitute a relationship depends on the context and is typically defined by the user.

Obviously, nesting is a very restrictive referencing mechanism. It constrains data sets to strict hierarchical relationships. Explicit associations, although favored in many data models, impose an implicit constraint on the data, too. They require that every relationship between entities has to be explicitly marked as one in the data before the relationship can be queried. In contrast, the value reference used in the relational model solely depends on the user's interpretation and can vary from context to context. In the relational model, the user explicitly expresses what constitutes a relationship, either with explicit foreign key constraints for base relations or join operations in queries.

2.2 Related Work

Over the decades, research and development in data management and computer science in general has created many data models and approaches to represent data. Obviously, we can concentrate only on the ones most prominently used for representing structured data. The baseline model for structured data serves as a reference. We consider a model as a structured data model if it encompasses at least the elements of the baseline model, namely: entity, entity domain, value, value domain, and technical type. Data models worth considering can be grouped in four main categories: (1) relational models, (2) software models, (3) document models, and (4) graph data models. Table 2.1 lists the data models grouped by the four categories. In the following, we briefly present and discuss these models with regard to their implicit constraints and which flexibility requirements they fulfill.

2.2.1 Relational Models

What we discuss as relational models here represent extensions of the traditional relational model as introduced by Codd [1970]. These extensions intend to free the relational model from one or more implicit constraints. Hence, these extended relational models allow additional flexibility compared to the pure relational model. Besides, all these extensions preserve the flexible value-based identity and value-based referencing of the relational model.

Extended NULL Semantic In the relational world, the NULL value takes an exceptional position among all values. NULL is the only single value that is separately interpreted and treated with a special logic. Generally, NULL serves as a placeholder for regular values and denotes that the actual value is not present. The actual value can be absent for a variety of reasons. For instance, ANSI/X3/SPARC [1975] lists in total 14 reasons, each suggesting a slightly different interpretation and handling

Table 2.1: Existing flexible data models.

Model category	Discussed examples
Relational	Extended NULL semantic [Bosak et al., 1962, Vassiliou, 1979] Interpreted column [Acharya et al., 2008, Foping et al., 2009] Interpreted record [Beckmann et al., 2006, Chu et al., 2007] Polymorphic table [Aulbach et al., 2011]
Software	Object orientation Role modeling [Steimann, 2000]
Document	XML [W3C, 2008, 2011a,b] JSON [Crockford, 2006] OEM [Papakonstantinou et al., 1995]
Tabular	Bigtable [Chang et al., 2006, 2008]
Graph	Property graph [Rodriguez and Neubauer, 2010] Neo4J [Neo Technology, 2013b] Freebase [Bollacker et al., 2008] RDF [W3C, 2004a] RDF w/ RDF Schema [W3C, 2004b]

of the NULL value. However, traditional relational database systems and the SQL standard [ISO/IEC, 2006] treat NULL values inconsistently [Date, 1984], not following a particular logical semantic. The most suitable interpretation sees NULL as an unknown value, i.e. an entity that is NULL in a given value domain still instantiates this value domain. The value only happens to be unknown. In contrast, entities with a generalized intention do not instantiate a given value domain. Here, the value does not exist. For instance, an entity representing a married person instantiates the value domain *wedding day*, even if the date is unknown. For an unmarried person, however, the date is not unknown; it does not exist. Hence, the entity representing the unmarried person does not instantiate the value domain *wedding day*. The extended NULL semantic distinguishes both interpretations [Bosak et al., 1962, Vassiliou, 1979]. This distinction allows the relational model to represent generalized entities and thereby removes one of its implicit constraints. The other four implicit constraints of the relational model remain.

Interpreted Column In the relational model, tuples instantiate only the columns of the table they are contained in. In other words, entities cannot instantiate value domains other than the ones associated with their entity domains. The interpreted column approach is an extension to the relational model that allows entities with a specialized intension [Acharya et al., 2008, Foping et al., 2009]. Relations that can be specialized contain an additional text column next to their core schema. Without any specialization the column remains empty. If a specialized entity is added to the

relation, the interpreted column approach represents the additional values of the entity in this column as text serialization. For each entity, the serialization contains the auxiliary values plus a reference to their value domains. The specific serialization technique used is independent of the principles of the approach. Usually, other data models with a defined text serialization, such as XML [W3C, 2008], JSON [Crockford, 2006] or CSV [Shafranovich, 2005], serve the purpose. In a database management system, the query processor has to de-serialize the interpreted column before evaluating query operations on the auxiliary values. Hence, the interpreted column approach is convenient for database management systems that offer direct query support for complex values in columns typed by a secondary data model. For instance, IBM DB2 supports XML columns [Cheng and Xu, 2000, Nicola and der Linden, 2005], which can be directly queried with XQuery [W3C, 2010a]. In such a case, the supported data model is strongly preferable for the serialization, because the query processor can directly leverage the existing query capabilities. Where such XML support exists, the interpreted column approach is appealing because it easily frees the relational model of an implicit constraint and allows specialized entities. However, it does not remove the other four implicit constraints of the relational model.

Interpreted Record The interpreted record concept [Beckmann et al., 2006, Chu et al., 2007] represents generalized and specialized entities in the relational model without explicitly representing the nonexistence of values or relying on a secondary data model. In the traditional relational model, the value domains that entities instantiate are implicitly defined via the entity domain, which constitutes the implicit constraints forbidding generalized and specialized entities. Instead, an interpreted record explicitly marks every value with a reference to the corresponding value domain. By this means the interpreted record omits NULL values as workaround representation of not instantiated value domains. Additionally, specialized entities can be represented without affecting the representation of the other entities in the same entity domain. Nevertheless, value domains remain strictly associated with entity domains. Hence, the interpreted record concept does not support independent value domains. Multifaceted entities and independent technical types are not supported either.

Polymorphic Table The polymorphic table concept [Aulbach et al., 2011] originates from the multi-tenancy domain [Jacobs, 2005, Jacobs and Aulbach, 2007]. In multi-tenant databases, tenants share the database schema, i.e. entity domains, value domains, and technical types or even data, i.e. entities and values. A crucial challenge in multi-tenant databases is that tenants want to be able to customize metadata and data even when it is shared [Lehner and Sattler, 2013]. Polymorphic tables [Aulbach et al., 2011] combine sharing and customization of metadata and data in a single concept. Generally, customization is the process of deriving a specialized table from a shared base table. In that sense, customization is similar to object inheritance (or specialization) [Currim and Ram, 2010]. The custom table of a tenant inherits schema and data of the table it customizes. Multiple customizations of the same base

table form an inheritance hierarchy, which is consolidated in one polymorphic table. Polymorphic tables allow the representation of specialized entities since these can be understood as a customization of their entity domain. In a limited way polymorphic tables also support multifaceted entities. Next to base tables, the concept allows the definition of extensions, which are named, predefined schema customizations. These extensions provide a means to add facets to an entity. Compared to full support for multifaceted entities, extensions remain limited, though. Extensions are always derived from a base table and cannot be arbitrarily combined across base tables. Support for generalized entities is not provided by the concept but could be easily added as a customization in a similar way polymorphic tables allow customizations to hide entities from a base table. Independent value domains and independent technical types are not supported.

Summarizing the extensions for the relational model, we can say that reasonable extensions exist to support generalized and specialized entities. We are not aware of extensions that add support for multifaceted entities, independent value domains, and independent technical types to the relational model.

2.2.2 Software Models

Software models originate from programming languages and other software development technologies. Generally, we see that software models consist of elements to structure operations, e.g. a function, and elements to structure data, e.g. a variable. The elements to structure data resemble a data model. In many software models, however, these elements are weaker than our baseline model for structured data. We focus our discussion on object orientation and role modeling. Both build on the notion of an object and encompass a dedicated association element to represent relationships between entities. Accordingly, they provide neither value-based object identity nor value-based referencing.

Object Orientation The notion of an object first appeared in the programming language LISP in the early 1960s [McCarthy et al., 1960, 1962]. In the late 1960s, Simula, which is considered the first object-oriented programming language, introduced objects, classes and methods as primary programming concepts [Dahl et al., 1970]. Since then, object-orientation has been repeatedly characterized, defined, and implemented in countless research work, programming languages, and development tools. Today, there is no clear definition of object-orientation. An analysis of 88 papers from nearly four decades of computing literature found *Inheritance*, *Object*, and *Class* to be the most fundamental concepts of object-orientation [Armstrong, 2006]. These are also the essential concepts allowing the use of object-orientation as a structured data model.

Objects have attributes. The set of attributes an object has is defined by the class the object is an instance of. With regard to the baseline model, objects resemble entities and classes resemble entity domains. With inheritance, classes can adopt the

structure from other classes, which is a restrictive means for representing multifaceted entities. Typically, inheritance is limited to a strictly hierarchical pass down of structure from the parent class to the child class. Even with multi-inheritance or concepts such as mixins [Moon, 1986, Bracha and Cook, 1990] or traits [Schärli et al., 2003], inheritance remains a relationship between classes; objects remain the instance of a single class. Hence, inheritance and related concepts to structure programming code do not offer full support for multifaceted entities. Furthermore, classes are strict blue prints for objects; generalization or specialization is not allowed for individual objects. Similarly, value domains are tied to classes and are not independent. Some programming languages offer independent technical types, in this context often referred to as weakly typed values. The concept of object-orientation does not define whether values are weakly or strongly typed.

Role Modeling Role modeling distinguishes entity domains in natural types and roles [Steimann, 2000]. Roles are entity domains that are founded and lack rigidity. Entities from a domain representing a role can only exist in relationship to other entities. When leaving this relation, an entity also leaves the domain but retains its identity. In contrast, natural types are entity domains that are rigid but lack foundation. Entities from a domain representing a natural type do not require a relation to other entities to exist but would lose their identity on leaving the domain. Role modeling allows multifaceted entities in the pre-defined form of roles. Dynamically, entities can start playing a role, i.e. join the corresponding entity domain, as well as stop playing a role, i.e. leave the domain. For each role it has to be modeled upfront which natural type can play the role, which does not realize the full flexibility we aim at with multifaceted entities. Role modeling does not contribute to the other flexibilities.

With inheritance and the notion of roles, software models offer limited support for multifaceted entities. Particularly the role concept allows the dynamic leaving and joining of entity domains. Nevertheless, the set of entity domains to which an entity can belong at runtime has to be determined upfront.

2.2.3 Document Models

Document models have been developed for representing documents, e.g. web pages. Typically, document models represent documents as a hierarchy of entities, where entities nest other entities. Nesting is the primary means of entity referencing. The identity of an entity solely or primarily depends on the position of the entity within the hierarchy. In consequence, document models offer neither value-based referencing nor value-based identity.

XML XML [W3C, 2008] is a markup language for documents. The textual data format intends to be machine-readable as well as readable by human. An XML document represents semi-structured data consisting of typically natural language

text and tags as markups to structure the text. Omitting the text, the tags can be used to structure data. Tags have a name and can contain attribute–value pairs, so that tags allow representing entities; the tag name denotes the entity domain and the attributes denote value domains. The structure of an XML document can be defined with a Document Type Definition (DTD) or another schema language, such as XML Schema [W3C, 2011a,b]. Both DTD and XML Schema associate value domains with the entity domains and specify technical types for the value domains. Well-formed XML documents are not required to follow the schema specifications of a DTD or an XML Schema; they must only satisfy syntactical rules given in the XML specification. Valid documents additionally conform to the rules of a DTD or an XML Schema. Consequently, we have to distinguish between well-formed documents and valid documents to assess XML regarding the supported flexibilities.

Obviously, well-formed XML documents have fewer implicit constraints and offer more flexibility. Without DTD or XML Schema, all value domains are per se independent from entity domains. Tags can have arbitrary attributes, which allows generalized entities as well as specialized entities. Still, tags exhibit only a single tag name, so that multifaceted entities are not supported. Technical types are not distinguished in well-formed XML.

Valid XML documents are stricter. DTD or XML Schema defines a schema for entity domains and these schemas are enforced. Nevertheless, it is possible to explicitly allow generalized entities and specialized entities in the schema for a given entity domain. There is also a limited support for multifaceted entities in the shape of extensions. Valid XML document do not allow independent value domains and independent technical types.

The nesting of tags presents the primary mechanism to relate entities in XML. Next to nesting, tags can cross-reference each other based on attribute values. In the DTD an attribute can be declared as IDREF. IDREF values must match the value of an ID attribute on some element in the XML document. The mechanism is similar to the relational foreign key concept. Compared to value-based referencing, however, it is limited because it prescribes the ID value as counterpart to the reference. XLink [W3C, 2010b] offers a third referencing mechanism for XML. XLink references XML elements with URIs, similar to links in HTML. Hence, an XLink resembles an explicit association and not a value-based reference.

JSON JavaScript Object Notation (JSON) [Crockford, 2006] is a very simple document model. Similar to XML, it is a textual data format design of a human-readable and portable representation of structured data. A JSON document consists of nested objects. Objects have attributes whose values can be strings, numbers, booleans, arrays of values, or even objects. Objects resemble the entities in our baseline model and attributes act as value domains. There is no equivalent for entity domains. Consequently, objects can have arbitrary attributes, but they are not identifiable in a logical group. Hence, the flexibility of the JSON objects comes at the price of functionality. JSON values have technical types, which are independent from

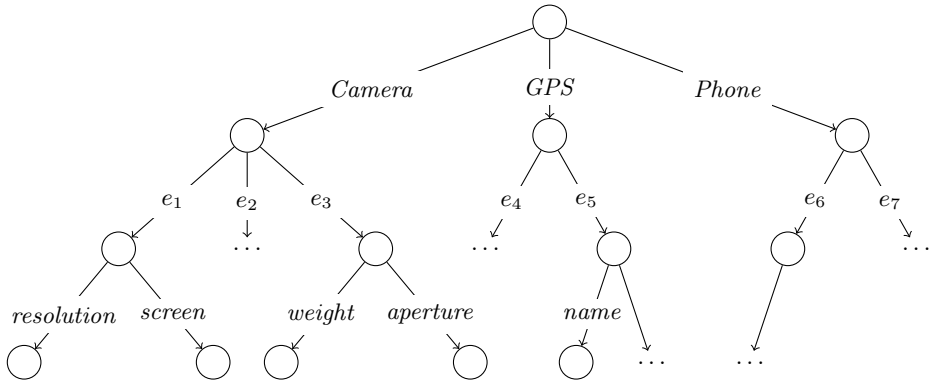


Figure 2.1: Example of structured data represented in OEM.

the value domains. The identity of objects is strictly positional. Next to the nesting there is no other referencing mechanism.

OEM The Object Exchange Model (OEM) [Papakonstantinou et al., 1995] shares some similarities with JSON but is about a decade older. It was developed in the 1990s for integrated access and data exchange between heterogeneous information sources. Each OEM data set resembles a tree of nested objects. OEM objects are fairly simple. They consist of a label, a data type, and a value. The data type can be an atom type, e.g. integer, float, string; or set. If the value is a set, it contains other OEM objects. Thereby, sets allow composing structures of higher order such as entities in OEM. OEM is more general then the baseline model. It can represent single values, sets or lists of values, single records, or with four levels of nesting, OEM can represent structured data as in the baseline model, as illustrated in Figure 2.1. Then, the topmost object represents the whole data set containing OEM objects, that each represents an entity domain. These second-level OEM objects are also typed as set and contain the OEM objects representing the entities of this domain, which in turn contain OEM objects representing the values. These four-level OEM objects are the leave nodes of the tree. Their label and their type correspond to the value domain and the technical type of the represented value, respectively. OEM does not distinguish the concepts of entity, entity domain, or value domain; in OEM they are all labels. In consequence, OEM is purely descriptive and very flexible. As in other document models, the identity of OEM objects is positional. For entity identity, however, the labels of the third-level OEM objects can also be used as identity. Nevertheless, the entity identity is not value-based. The same holds for referencing.

Document models offer more flexibility than most relational systems or software models. However, most of their flexibility comes from completely omitting entity domains or a relation between entity domains and value domains. Where document models have schema information, such as DTD or XML Schema for XML documents,

they are similarly strict. Additionally, with nesting, document models exhibit a very restrictive primary referencing mechanism.

2.2.4 Tabular Models

A tabular data model also organizes data in tables like the relational data model but differs so significantly in its design that it cannot be considered relational. The data model of Google's Bigtable system [Chang et al., 2006, 2008] defined the category of tabular data models. To fulfill the needs of its large web applications, Google strictly designed the Bigtable system and its data model for scale out of petabytes of data across thousands of servers. The Bigtable data model has been implemented in various systems, with Cassandra [Lakshman and Malik, 2010] and Apache HBase [Apache Software Foundation, 2013] being the most successful ones. Becaused of its success, it has remained the only model of its kind that draws considerable attention.

Bigtable organizes data in large, distributed, sparse tables. The columns of such a table are grouped in column families. Hence, a table in Bigtable resembles a complete database rather than a single table in the relational sense or a single entity domain in the sense of our baseline model. Column families take the role of entity domains, instead. Nevertheless rows, which resemble entities, can stretch across multiple column families, so that the Bigtable data model supports multifaceted entities. Although a column family can define a set of columns that rows should instantiate if they appear in the column family, rows are free to instantiate any column in a column family. Consequently, Bigtable allows arbitrarily generalized and specialized entities. Similarly, the technical type of a value can differ from the technical type a column may define, so that the Bigtable model also supports independent technical types. Independent value domains are not supported, though. In Bigtable, columns with the same name can appear in different column families, but they are not the same column, as columns cannot be identified independently from their column family. Rows are identified by a user-given row key. Value-based identity is not supported. Referencing between rows works in a value-based way like the relational foreign key. However, there is no join operation in Bigtable, and a row can only be retrieved by its row key. Accordingly, Bigtable foreign keys are limited to row keys. References targeting any other value than the row key can be stored but not retrieved.

The Bigtable model offers considerably more flexibility than the relational data model. Because of its focus on scalability, however, it lacks flexibility regarding entity identity and entity referencing.

2.2.5 Graph Models

Graph data models build on the mathematical definition of a graph. They represent data as vertices and edges, where vertices represent entities and edges represent relationships, i.e. references to other entities. In all graph models, entities have an object identity and edges are an explicit representation of references. Consequently, graph models do not allow value-based identity or value-based referencing. In this

respect, graph models are very similar to software models. The essential difference to software models is that graph models follow the principle of schema-comes-second. While graph models emphasize the representation of data, software model focus on the modeling of schema. The different focus gives graph models a descriptive nature.

In practice, graph models differ in how data is represented in a graph. Beside vertices and edges, graphs can have labels and attribute–value pairs attached to the vertices and even to the edges. [Rodriguez and Neubauer, 2010] distinguishes nine types of graphs. Most prominent in data management are the property graph and the RDF graph. In any case, the pure graph models do not provide schema specifications. Some systems or standards overcome that by extending their underlying graph model with additional elements.

Property Graph The property graph is probably the most common type of graph used in graph databases, e.g. SAP HANA [Rudolf et al., 2013]. As a directed, labeled, attributed multi-graph, the property graph is a structure powerful enough to embody all the other eight graph types [Rodriguez and Neubauer, 2010]. Graphs of this type have labels and attribute–value pairs for vertices as well as edges. For structured data, vertices represent entities, vertex labels serve as entity domains and attribute–value pairs (a.k.a. properties) express value domains and values. Technical types are not specified in the property graph model. Similar to well-formed XML, property graphs do not include schema specification. Likewise, value domains are per se independent from entity domains and entities can be arbitrarily generalized and specialized. This allows property graphs a great deal of flexibility. On the downside, they are constrained to a single label per vertex, which prohibits multifaceted entities.

Neo4j Neo4j [Neo Technology, 2013b] is currently one of the most popular graph database engines. Its proprietary data model is a modified version of the property graph model, inheriting most of the property graph's flexibility. In its first versions, Neo4j lacked vertex labels and thereby entity domains. With version 2.0, though, Neo4j introduces vertex labels in a more powerful fashion than node labels in property graphs [Rathle, 2013, Neo Technology, 2013a, Hunger, 2013]. In contrast to the property graph, Neo4j allows multiple labels per vertex, which paves the way for support of multifaceted entities. Later, the label should be also used for explicit schema constraints. As Neo4j is written in Java, values are technically typed but independent from the value domain.

Freebase Freebase [Bollacker et al., 2008] is a public graph database operated by Google. Its proprietary data model borrows from the property graph model but lacks vertex labels and edge properties. Freebase represents entity domains with a defined vertex property named *type*. Similar to JSON, values can be arrays, allowing Freebase to represent multifaceted entities – a feature heavily used in the Freebase schema. In the Freebase schema, every entity domain lists a set of value domains and their expected type. Value domains are bound to the entity domain at which they are

listed; they cannot exist independently. Entities can generalize the intension of their entity domains; specialization is not supported. The expected type of a value domain is either a technical type in the case of a value or an entity domain in the case of a reference. The Freebase engine enforces expected types, so that it does not support independent technical types.

RDF The Resource Description Framework (RDF) [W3C, 2004a] is widely used for conceptual modeling and annotating structured information to web resources particularly in the context of the so called semantic web [Berners-Lee et al., 2001, Shadbolt et al., 2006]. RDF represents data statements about entities. Statements are stored as subject-predicate-object triples; all triples can be interpreted as a labeled graph. Accordingly, an RDF graph is a considerably simpler data model than a property graph. Nevertheless, it is powerful enough to represent structured data. The subject of a triple is typically an entity; the predicate is a value domain and the object is a value or a reference. In RDF, everything except values is identified and expressed by URIs [Berners-Lee et al., 2005]. In each triple, subject and predicate are URIs, while the object is a URI only if it represents a reference. Similar to JSON, RDF lacks entity domains. Consequently, entities can have arbitrary attributes and value domains are independent, but means to identify logical groups of entities are missing. RDF knows the most basic technical types and allows them independently from the predicate in a triple.

RDF with RDF Schema RDF Schema [W3C, 2004b] is a set of predicates and objects that allow the description of a schema for RDF data in RDF itself. One of the main predicates is *rdf:type*, which assigns an entity to a class. Classes represent either entity domains or value domains in RDF Schema; the *rdf:type* of the class distinguishes between both. Value domains define an *rdfs:domain*, which links them to entity domains, and an *rdfs:range*, which specifies the technical type of values or the target entity domain for references. RDF entities do not have to comply with their classes; generalization and specialization of entities is allowed. Generally, RDF allows declaring the same predicate multiple times for the same object. Only the complete subject-predicate-object triple has to be unique. This has three essential consequences for the flexibility of RDF Schema. First, multifaceted entities are possible by simply assigning multiple classes to an entity. Second, value domains are independent from the entity domains since they can declare multiple entity domains as their *rdfs:domain*. Third, value domains can specify multiple technical types. Although this allows weakly typed values in some sense, technical types are not completely independent since types that are used for a value domain have to be specified in the schema.

One main reason for the current popularity of graph models is, as we have seen, the flexibility they provide. The other main reason is that graph models center around representing relationships between entities, which is what is primarily needed in many modern applications. However, the fixed object identity and the explicit

representation of relationships are also a drawback since they limit the flexibility in interpreting the data. Another drawback particularly of the property graph model and its descendant used in SAP HANA, Neo4j, and Freebase is, that they lack any kind of standardization currently.

2.2.6 Summary

As a summary, Table 2.2 shows which flexibilities each of the discussed data models allows. We can see that none of the discussed models is completely constraint-free and provides the flexibilities and characteristics provided by FRDM. Graph models, particularly as in Neo4j, are free of implicit constraints regarding entity domains, value domains and technical types, while the relational approaches are the only ones to offer value-based identity and value-based references. FRDM integrates the level of flexibility graph models provide with value-based identity and value-based references, as indicated in Table 2.2, in a super-relational fashion. In the next section, we present the flexible relational data model FRDM.

2.3 FRDM

FRDM is a super-relational data model for structured data. It is free of the relational inflexibilities but keeps the relational flexibilities. The most prominent feature of FRDM is that it separates the functionality of data representation, data processing, and constraints. Data representation and data processing are realized in separate, dedicated concepts. We detail the data representation of FRDM in Section 2.3.1 and discuss data processing in Section 2.3.2. Schema constraints are realized as explicit constraints outside of the core data model in the constraint framework FRDM-C, which is presented in Section 2.4.

2.3.1 Data Representation

The data representation of FRDM builds on four concepts. The central concept is the *tuple*:

Tuple A tuple is the central concept of the flexible relational data model and represents an entity. A tuple consists of values. Each value is part of a value domain and is encoded according to a technical type.

The concepts *entity domain*, *value domain*, and *technical type* describe the data represented in tuples and provide logical data handles:

Entity domain Entity domains are logical data handles allowing to distinguish logical groups of tuples within a database. Tuples belong to at least one entity domain and may belong to multiple entity domains, so that domains can intersect each other.

Table 2.2: Flexible data models vs. requirements.

	Generalized entities	Specialized entities	Multi-faceted entities	Independent value domains	Independent technical types	Value-based identity	Value-based referencing
Pure relational						✓	✓
Extended NULL semantic	✓					✓	✓
Interpreted column		(✓)[1]				✓	✓
Interpreted record	✓	✓				✓	✓
Polymorphic table	(✓)[2]	✓	(✓)[3]			✓	✓
FRDM w/ FRDM-C	✓	✓	✓	✓	✓	✓	✓
Object orientation			(✓)[4]		(–)[6]		
Role modeling			(✓)[5]		(–)[6]		
XML, well-formed	✓[7]	✓[7]		✓[7]	(✓)[8]		
XML, valid	(✓)[1]	(✓)[1]	(✓)[3]				(✓)[9]
JSON	(✓)[10]	(✓)[10]	(✓)[10]	(✓)[10]	✓[11]		
OEM	(✓)[10]	(✓)[10]	(✓)[10]	(✓)[10]	✓[11]		
Bigtable	✓	✓	✓		✓		(✓)[12]
Property graph	✓[7]	✓[7]		✓[7]	(–)[6]		
Neo4J	✓	✓	✓	✓	✓[11]		
Freebase	✓		✓				
RDF	(✓)[10]	(✓)[10]	(✓)[10]	(✓)[10]	✓		
RDF w/ RDF Schema	✓	✓	✓	✓	(✓)[1]		

[1] must be explicitly allowed in schema [2] could be added [3] extensions [4] inheritance [5] roles [6] not specified [7] no relation between entity domains and value domains [8] no technical types [9] secondary value-based referencing mechanism [10] no entity domains [11] no relation between value domains and technical types [12] limited by object-id-based entity identity

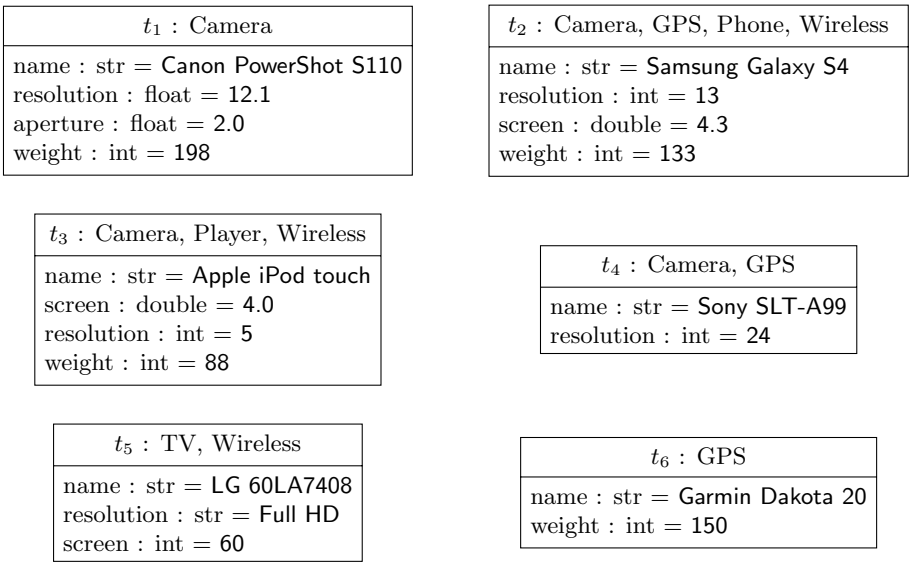


Figure 2.2: Example entities representing electronic devices.

Value domain Value domains are logical data handles allowing to distinguish values within a tuple. Each tuple can instantiate each value domain only once.

Technical type Technical types determine the physical representation of values. Value operations such as comparisons and arithmetic are defined at the level of technical types.

Formally, a flexible relational database is a septuple $(\mathbb{D}, \mathbb{A}, \mathbb{T}, \mathbb{E}, f_s, f_t, f_m)$. The payload data $\mathbb{D}$ is a set of tuples. A tuple is an ordered set of values $t = [v_1, \ldots, v_m]$. Let $\mathbb{A}$ be the set of all value domains available in the database. Then the tuple schema function $f_s : \mathbb{D} \to \mathcal{P}(\mathbb{A})$ denotes the schema of each tuple, i.e. the set of value domains a tuple instantiates. $f_s(t) = [A_1, \ldots, A_m]$ if t instantiates the value domains $A_1, \ldots, A_m$ so that $t \in A_1 \times \cdots \times A_m$. For convenience, we denote with $t[A] = v$ that tuple t instantiates value domain A with value v. $\mathbb{T}$ is the set of all available technical types T. The typing function $f_t : \mathbb{D} \times \mathbb{A} \to \mathbb{T}$ shows the encoding of values, with $f_t(t, A) = T$ if the value $t[A]$ is encoded according to the technical type T. Finally, $\mathbb{E}$ is the set of all available entity domains E, while the membership function $f_m : \mathbb{D} \to \mathcal{P}(\mathbb{E})$ denotes which tuples belong to these domains. $f_m(t) = \{E_1, \ldots, E_k\}$ if t belongs to the entity domains $E_1, \ldots, E_k$.

As an example, Figure 2.2 shows six entities in UML-object-diagram-like notation. The entities represent electronic devices as they could appear in a product catalog. Note that this small example exploits all the flexibilities of FRDM. All six entities are self-descriptive and have their individual set of attributes. The order of the attributes within an entity differs, too. Entities t_2 to t_5 belong to multiple entity domains. Attributes, such as *name*, appear independently from entity domains. The technical typing of values, for instance of the attribute *resolution*, varies independently of

the attribute. In the flexible relational data model these six entities can be directly represented as follows.

Tuples:

$$\mathbb{D} = \left\{ \begin{array}{l} t_1 = [\text{Canon PowerShot S110}, 12.1, 2.0, 198]\,, \\ t_2 = [\text{Samsung Galaxy S4}, 13, 4.3, 133]\,, \\ t_3 = [\text{Apple iPod touch}, 4.0, 5, 88, \text{Yellow}]\,, \\ t_4 = [\text{Sony SLT-A99}, 24]\,, \\ t_5 = [\text{LG 60LA7408}, \text{Full HD}, 60]\,, \\ t_6 = [\text{Garmin Dakota 20}, 150] \end{array} \right\}$$

Schema elements:

$$\begin{aligned} \mathbb{A} &= \{\mathit{aperture}, \mathit{name}, \mathit{resolution}, \mathit{screen}, \mathit{weight}\} \\ \mathbb{T} &= \{\mathit{float}, \mathit{int}, \mathit{str}\} \\ \mathbb{E} &= \{\mathit{Camera}, \mathit{GPS}, \mathit{Player}, \mathit{Phone}, \mathit{TV}, \mathit{Wireless}\} \end{aligned}$$

Schema function:

$$f_s = \left\{ \begin{array}{l} t_1 \rightarrow [\mathit{name}, \mathit{resolution}, \mathit{aperture}, \mathit{weight}]\,, \\ t_2 \rightarrow [\mathit{name}, \mathit{resolution}, \mathit{screen}, \mathit{weight}]\,, \\ t_3 \rightarrow [\mathit{name}, \mathit{screen}, \mathit{resolution}, \mathit{weight}]\,, \\ t_4 \rightarrow [\mathit{name}, \mathit{resolution}]\,, \\ t_5 \rightarrow [\mathit{name}, \mathit{resolution}, \mathit{screen}]\,, \\ t_6 \rightarrow [\mathit{name}, \mathit{weight}] \end{array} \right\}$$

Typing function:

$$f_t = \left\{ \begin{array}{l} (t_1, \mathit{name}) \rightarrow \mathit{str}, (t_1, \mathit{resolution}) \rightarrow \mathit{float}, (t_1, \mathit{aperture}) \rightarrow \mathit{float}, \ldots \\ (t_2, \mathit{name}) \rightarrow \mathit{str}, (t_2, \mathit{resolution}) \rightarrow \mathit{int}, (t_2, \mathit{screen}) \rightarrow \mathit{double}, \ldots \\ (t_3, \mathit{name}) \rightarrow \mathit{str}, (t_3, \mathit{screen}) \rightarrow \mathit{double}, (t_3, \mathit{resolution}) \rightarrow \mathit{int}, \ldots \\ (t_4, \mathit{name}) \rightarrow \mathit{str}, (t_4, \mathit{resolution}) \rightarrow \mathit{int}, \\ (t_5, \mathit{name}) \rightarrow \mathit{str}, (t_5, \mathit{resolution}) \rightarrow \mathit{str}, (t_5, \mathit{screen}) \rightarrow \mathit{int}, \\ (t_6, \mathit{name}) \rightarrow \mathit{str}, (t_6, \mathit{weight}) \rightarrow \mathit{int} \end{array} \right\}$$

Membership function:

$$f_m = \left\{ \begin{array}{l} t_1 \rightarrow \{\mathit{Camera}\}\,, \\ t_2 \rightarrow \{\mathit{Camera}, \mathit{GPS}, \mathit{Phone}, \mathit{Wireless}\}\,, \\ t_3 \rightarrow \{\mathit{Camera}, \mathit{Player}, \mathit{Wireless}\}\,, \\ t_4 \rightarrow \{\mathit{Camera}, \mathit{GPS}\}\,, \\ t_5 \rightarrow \{\mathit{TV}, \mathit{Wireless}\}\,, \\ t_6 \rightarrow \{\mathit{GPS}\} \end{array} \right\}$$

2.3.2 Data Processing

For data processing, FRDM builds on the well-known concept of a *relation*. It allows to process tuples in a relational manner:

Relation Relations serve as central processing containers for tuples. FRDM queries operate on relations; query operations have relations as input and produce relations as output. Different from the traditional relational data model, with FRDM the tuples in a relation determine the schema of the relation. Each value domain instantiated by at least a single tuple in the relation is part of the relation's schema.

Let t be a tuple in relation R, then R has the schema

$$S_R = \bigcup_{t \in R} f_s(t)$$

so that $S_R \subseteq \mathbb{A}$. A relation R with schema S_R does not have to instantiate each tuple in every value domain, rather it is

$$R \subseteq \bigcup_{S_i \in \mathcal{P}(S_R) \setminus \emptyset} \left(\bigtimes_{A \in S_i} A \right).$$

In other words, tuples may only instantiate a subset of a relation's schema, with the exception of the empty set. While $t[A] = v$ denotes that tuple t instantiates value domain A with value v, $t[A] = \nexists$ indicates that tuple t does not instantiate value domain A.

Mass operations address tuples by means of entity domains. Hence, each entity domain denotes a relation containing all tuples that belong to this domain. Specifically, an entity domain E denotes a relation R so that $E \in f_m(t)$ holds for all $t \in R$. In the following, we refer to a relation representing tuples of domain E simply as E where unambiguously possible. Figure 2.3 shows two relations in the electronic device example denoted by the entity domains *Camera* and *GPS*, respectively.

The well-known relational operators (selection, projection, union, difference, cross-product) are directly applicable to FRDM relations. However, the descriptive nature of a FRDM relation requires two minor modifications to the semantic of the relational operators. First, the logic of selection predicates and projection expressions has to consider that value domains may not be instantiated by a tuple. An appropriate evaluation function for such predicates and expressions is described in [Vassiliou, 1979]. In a nutshell, tuples that do not instantiate a value domain used in a selection predicate are not applicable to the predicate and do not qualify. Tuples that do not instantiate a value domain used in a projection expression do not instantiate the value domain newly defined by the expression. Second, all operations have a strictly tuple-oriented semantic, i.e. the schema of the relation resulting from an operation is solely determined by the qualifying tuples. In consequence, the schema resulting from a selection can differ from the schema of the input relation. More specifically, the

Relation *GPS*

name	resolution	screen	weight
Samsung Galaxy S4	13	4.3	133
Sony SLT-A99	24	∄	∄
Garmin Dakota 20	∄	∄	150

Relation *Camera*

name	resolution	aperture	weight	screen
Canon PowerShot S110	12.1	2.0	198	∄
Samsung Galaxy S4	13	∄	133	4.3
Apple iPod touch	5	∄	88	4.0
Sony SLT-A99	24	∄	∄	∄

Figure 2.3: Relations denoted by entity domains.

Relation $Camera \cap GPS$

name	resolution	screen	weight
Samsung Galaxy S4	13	4.3	133
Sony SLT-A99	24	∄	∄

Relation $\sigma_{aperture \leq 2.8} Camera$

name	resolution	aperture	weight
Canon PowerShot S110	12.1	2.0	198

Figure 2.4: Relations resulting from relational operators.

resulting schema of a selection is equal to or a subset of the input schema depending on which tuples qualify, so that $S_{\sigma_P(A)} \subseteq S_A$. Likewise, the schemas of the operand relations do not matter for set operations. Tuples are equal if they instantiate the same value domains with equal values. In the case of a union the schema results from uniting the schemas of the operands, so that $S_{A \cup B} = S_A \cup S_B$. In the case of a set difference the resulting schema is equal to or a subset of the left operand's schema, again, depending on which tuples qualify, so that $S_{A \setminus B} \subseteq S_A$. Derived operators, such as join or intersection, are affected similarly.

As an example, Figure 2.4 shows two relations resulting from relational operators. Relation $Camera \cap GPS$ is the intersection of the relations *Camera* and *GPS* as shown in Figure 2.3. Relation $\sigma_{aperture \leq 2.8} Camera$, obviously results from a selection on value domain *aperture* of the *Camera* relation.

2.4 FRDM-C

FRDM-C is a flexible constraint framework meant to accompany FRDM. The flexibility of FRDM originates from its lack of implicit constraints. Nevertheless, constraints are a powerful feature if their effect is desired by the user. For the user, constraints are the primary means to have the data management system to obtain and maintain data quality. Each constraint is a proposition about the data in the database. Data either complies to or violates this proposition, i.e. every constraint categorizes the data into two disjoint subsets. It is up to the user how to utilize this categorization. At the least, constraints inform about which data is compliant and which is violating. At the most, constraints prohibit data modifications that would result in violating data. Constraints present themselves as additional schema objects, attached to the schema elements of the data model. The user can add and remove constraints at any time.

Formally, constraints take the general form of a triple (q, c, o). q is the qualifier; c is the condition compliant data has to fulfill; o is the effect (or the outcome) the constraint will have. The qualifier determines to which tuples the constraint applies. It is either an entity domain $E_q \in \mathbb{E}$, a value domain $A_q \in \mathbb{A}$, or a pair of both (E_q, A_q). Correspondingly, a constraint applies to all tuples t with $E_q \in f_m(t)$, with $A_q \in f_s(t)$, or with $(E_q, A_q) \in f_m(t) \times f_s(t)$, respectively. We denote the set of tuples a constraint C applies to as $\mathbb{D}_C$. Conditions are either tuple conditions or key conditions, depending on whether they affect individual tuples or groups of tuples. The effect determines the result of the operations that lead to violating data and what happens to the violating data itself. In the following, we will detail conditions and effects.

2.4.1 Conditions

The first group of conditions is tuple conditions. Tuple conditions restrict data at the level of individual tuples, e.g. by mandating to which entity domains a tuple can belong. Formally, a tuple condition is a function $c : \mathbb{D} \rightarrow \{\top, \bot\}$. Then, $\mathbb{D}_C^{\top} = \{t \mid t \in \mathbb{D}_C \wedge c(t)\}$ are the complying tuples and $\mathbb{D}_C^{\bot} = \{t \mid t \in \mathbb{D}_C \wedge \neg c(t)\}$ are the violating tuples. Tuple conditions are:

Entity domain condition An entity domain condition requires tuples $t \in \mathbb{D}_C$ to belong to an entity domain E_c so that $E_c \in f_m(t)$. We denote a specified entity domain condition as *entity-domain*(E_c).

Value domain condition A value domain condition requires tuples $t \in \mathbb{D}_C$ to instantiate a value domain A_c so that $A_c \in f_s(t)$. We denote a specified value domain condition as *value-domain*(A_c).

Technical type condition A technical type condition limits values of tuples $t \in \mathbb{D}_C$ in value domain A_c to a specified technical type T_c so that $T_c = f_t(t, A_c)$. We denote a technical type condition as *tech-type*(A_c, T_c).

Value condition A value condition requires values of tuples $t \in \mathbb{D}_C$ in value domain A_c to fulfill a specified predicate p so that $p(t[A_c])$ holds. We denote a value condition as $\mathit{value}(A_c, p)$.

The second group of conditions is key conditions. Key conditions restrict data at the level of tuple groups. Formally, a key condition is a function $c : \mathcal{P}(\mathbb{D}) \to \{\top, \bot\}$. Key conditions are:

Unique key condition A unique key condition requires tuples to instantiate a set of value domains $\mathbb{A}_K \subseteq \mathbb{A}$ uniquely so that $t_i[\mathbb{A}_K] \neq t_j[\mathbb{A}_K]$ holds for all $t_i, t_j \in \mathbb{D}_C$ with $t_i \neq t_j$. As a result, all complying tuples are unambiguously identifiable on $\mathbb{A}_K$. We denote a unique key condition as $\mathit{unique\text{-}key}(\mathbb{A}_K)$.

Foreign key condition A foreign key condition requires tuples to instantiate value domains $\mathbb{A}_F \subseteq \mathbb{A}$ with values referencing at least one tuple on value domains $\mathbb{A}_R \subseteq \mathbb{A}$ so that for every $t_F \in \mathbb{D}_C$ there is one $t_R \in \mathbb{D}_R$ such that $t_F[\mathbb{A}_F] = t_R[\mathbb{A}_R]$. Similarly to $\mathbb{D}_C$, the set of referenceable tuples $\mathbb{D}_R \subseteq \mathbb{D}$ is identified by either an entity domain $E_R \in \mathbb{E}$, a value domain $A_R \in \mathbb{A}$, or a pair of both (E_R, A_R). We denote a foreign key condition as $\mathit{foreign\text{-}key}(\mathbb{A}_F, \mathbb{A}_R, m_R)$ where q_R is the qualifier of $\mathbb{D}_R$.

If a group of tuples does not fulfill a key condition not all tuples of the group are considered violating. We have to distinguish two cases. Consider a database with tuples representing authors and a unique key constraint requiring unique author names, as shown in Figure 2.5. In the first case, a constraint already exists in the database and a modification of tuples results in a violation. Here only the modified tuples become violating tuples. In the example shown in the figure, the database contains two tuples and the unique constraint. Some operation inserts a third tuple, which applies to the constraint so that the group of all author tuples violates the constraint's condition. As result, the set of violating tuples $\mathbb{D}_C^{\bot}$ encompasses the inserted tuples t_3. In the second case, the constraint is added to the database and the tuples already existing in the database violate this constraint. Here the smallest subset of tuples that violates the condition becomes the set of violating tuples, i.e. $\mathbb{D}_C^{\bot}$ is the set of violating tuples for a constraint C if $c(\mathbb{D}_C^{\bot}) = \bot$, $c(\mathbb{D} \setminus \mathbb{D}_C^{\bot}) = \top$, and $\left|\mathbb{D}_C^{\bot}\right|$ is minimal. In the example shown in the figure, the minimal set of tuples violating the unique constraint is $\{t_2, t_3\}$, so they become the violating tuples for the constraint.

All conditions can be negated in a constraint. Negation swaps the set of violating tuples with the set of complying tuples. For instance, the negated entity domain condition $\neg\mathit{entity\text{-}domain}(E_c)$ prohibits the entity domain E_c instead of requiring it. Given two constraints $C = (q, c, o)$ and $C' = (q, \neg c, o)$, it holds that $\mathbb{D}_{C'}^{\top} = \mathbb{D}_C^{\bot}$ and $\mathbb{D}_{C'}^{\bot} = \mathbb{D}_C^{\top}$. For any kind of constraint, the set of voilating tuples $\mathbb{D}_C^{\bot}$ is crucial for the effect of the constraint.

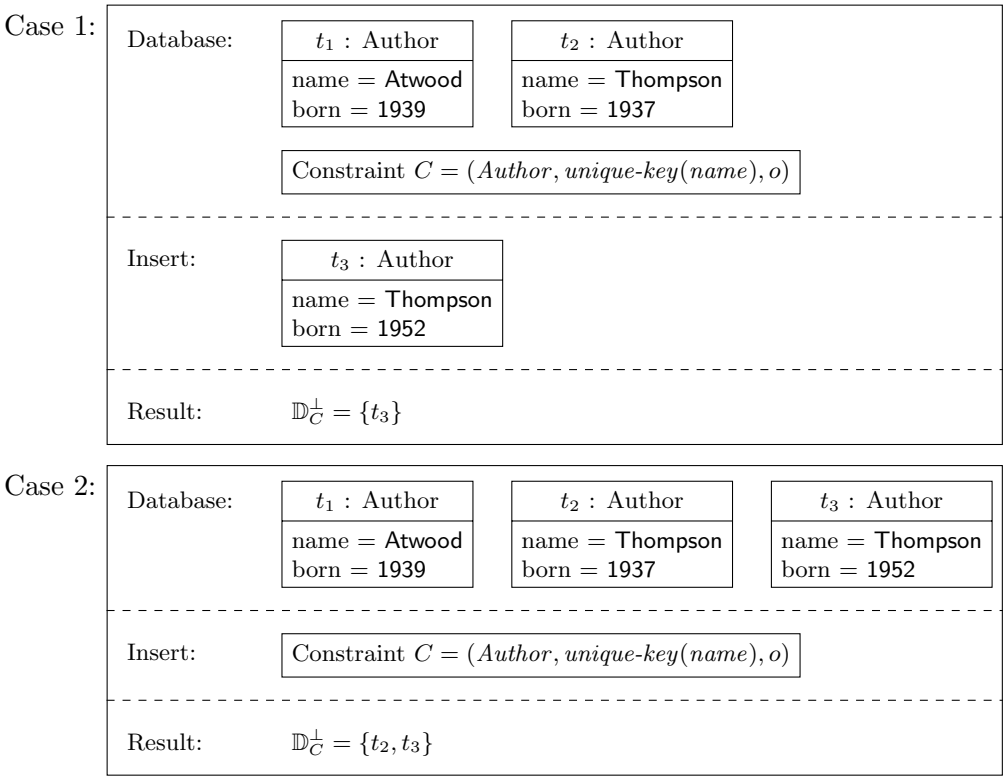


Figure 2.5: Set of violating tuples for a unique key condition

2.4.2 Effects

We distinguish four types of effects constraints can have. They vary in the rigor the constraint will exhibit.

Informing Allows all operations. The complying tuples and the violating tuples can be queried by using the constraint as a query predicate.

Warning Allows all operations and issues a warning upon operations that lead to violating tuples. The creation of the constraint results in a warning about already existing violating tuples.

Hiding Allows all operations and issues a warning upon operations that lead to violating tuples, and hides violating tuples from all other operations. The creation of the constraint results in hiding already existing violating tuples, except for operations that explicitly request to see violating tuples by using the constraint as predicate.

Prohibiting Prohibits operations that lead to violating tuples and issues an error. The creation of the constraint is prohibited in the case of already existing violating tuples.

2.5 Super-Relational Nature of FRDM

The presented flexible relational data model is a superset of the traditional relational model. Traditional relations can be represented directly in the flexible model without any change. A relational database is a septuple $(\mathbb{D}, \mathbb{A}, \mathbb{T}, \mathbb{R}, f_\sigma, f_\theta, f_\mu)$, where $\mathbb{R}$ is the set of relations, $\mathbb{A}$ is the set of domains, $\mathbb{T}$ is the set of technical types, $\mathbb{D}$ is the set of tuples, f_σ is the schema function $\mathbb{R} \to \mathcal{P}(\mathbb{A})$, f_θ is the typing function $\mathbb{A} \to \mathbb{T}$, and f_μ is the membership function $\mathbb{D} \to \mathbb{R}$. The corresponding flexible relational database is $(\mathbb{D}, \mathbb{A}, \mathbb{T}, \mathbb{E}, f_s, f_t, f_m)$ with

$$\begin{aligned}
\mathbb{E} &= \{\textit{name-of}(R) \mid R \in \mathbb{R}\} \\
f_s &= \{t \to f_\sigma(f_\mu(t)) \mid t \in \mathbb{D}\} \\
f_t &= \{(t, A) \to f_\theta(A) \mid A \in f_\sigma(f_\mu(t)) \wedge t \in \mathbb{D}\} \\
f_m &= \{t \to \{\textit{name-of}(f_\mu(t))\} \mid t \in \mathbb{D}\} \quad .
\end{aligned}$$

For each relation, we get an entity domain with the name of the relation. The schema of each tuple can be derived from the schema of the relation to which the tuple belongs in the relational model, so that $f_s(t) = f_\sigma(f_\mu(t))$. Additionally, the relational schema provides the technical type of a value. Since the relational data model does not allow independent technical types, the technical type is only determined by the attribute, so that $f_t(t, A) = f_\theta(A)$. Finally, each tuple belongs to only a single entity domain, specifically to the entity domain with the name of the tuple's relation, so that $f_m(t) = \textit{name-of}(f_\mu(t))$.

To emulate the implicit constraints of the relational model (cf. Section 2.1.1) the flexible relational database has to be supplemented with explicit constraints. For each relation $R \in \mathbb{R}$ we add the following prohibitive constraints:

- Entity domains have to mutually exclude each other, so that tuples can be only part of one entity domain. This can be achieved with constraints of the form $(\textit{name-of}(R), \neg\textit{entity-domain}(E), \textit{prohibiting})$ where $\textit{name-of}(R) \neq E$ and $R \in \mathbb{R}$.
- Entity domains prescribe the value domains of their corresponding relation. This can be achieved with constraints of the form $(\textit{name-of}(R), \textit{value-domain}(A), \textit{prohibiting})$ for each value domain $A \in f_\sigma(R)$ and each relation $R \in \mathbb{R}$.
- Entity domains forbid all other value domains. This can be achieved with constraints of the form $(\textit{name-of}(R), \neg\textit{value-domain}(A'), \textit{prohibiting})$ for each value domain $A' \notin f_\sigma(R)$ and each relation $R \in \mathbb{R}$.
- Value domains prescribe the technical type as defined by the corresponding relation. This can be achieved with constraints of the form $(A, \textit{tech-type}(A, f_\theta(A)), \textit{prohibiting})$ for each value domain $A \in f_\sigma(R)$ and each relation $R \in \mathbb{R}$.

As an example consider the two traditional relations shown in Figure 2.6. To emulate their relational rigidity we need the following 17 prohibitive constraints assuming no other entity domains and value domains are present in the database.

Relation *Author*

name : str	born : int
Margaret Atwood	1939
Hunter S. Thompson	1937
José Saramago	1922

Relation *Camera*

name : int	resolution : int	weight : int
Canon PowerShot S110	12	198
Samsung Galaxy S4	13	133
Sony SLT-A99	24	812

Figure 2.6: Two relations in the relational data model.

- Entity domains have to mutually exclude each other:
  - $(Author, \neg entity\text{-}domain(Camera), prohibiting)$
  - $(Camera, \neg entity\text{-}domain(Author), prohibiting)$
- Entity domains prescribe the value domains of their corresponding relation:
  - $(Author, value\text{-}domain(Author.name), prohibiting)$
  - $(Author, value\text{-}domain(Author.born), prohibiting)$
  - $(Camera, value\text{-}domain(Camera.name), prohibiting)$
  - $(Camera, value\text{-}domain(Camera.resolution), prohibiting)$
  - $(Camera, value\text{-}domain(Camera.weight), prohibiting)$
- Entity domains forbid all other value domains:
  - $(Author, \neg value\text{-}domain(Camera.name), prohibiting)$
  - $(Author, \neg value\text{-}domain(Camera.resolution), prohibiting)$
  - $(Author, \neg value\text{-}domain(Camera.weight), prohibiting)$
  - $(Camera, \neg value\text{-}domain(Author.name), prohibiting)$
  - $(Camera, \neg value\text{-}domain(Author.born), prohibiting)$
- Value domains prescribe the technical type as defined by the corresponding relation:
  - $(Author.name, tech\text{-}type(Author.name, str), prohibiting)$
  - $(Author.born, tech\text{-}type(Author.born, int), prohibiting)$
  - $(Camera.name, tech\text{-}type(Camera.name, str), prohibiting)$
  - $(Camera.resolution, tech\text{-}type(Camera.resolution, int), prohibiting)$
  - $(Camera.weight, tech\text{-}type(Camera.weight, int), prohibiting)$

2.6 Implementation of FRDM

The FRDM data model is positioned as a flexible descendant of the relational model. Therefore it is suitable to be implemented within the existing and established relational database system architecture. In this section, we briefly discuss how this can be done. The characteristics of FRDM require four main changes to existing relational database system code.

First, plan operators and query processing have to be adapted to handling descriptive relations. More specifically, plan operators must reflect the adapted semantic of their logical counterparts. Logically, operators have to remove value domains from the schema of a relation if no tuple instantiates them. With a tuple-at-a-time processing model, this orphaned value domain elimination is a blocking operation, since the system can determine the schema only after all tuples are processed. Implicit duplicate elimination is similarly impractical and thus was not implemented in relational database systems. Likewise a practical solution for the elimination of orphaned value domains is that plan operators determine the schema of the resulting operation as narrowly as they safely can before the actual tuple processing and live with possible orphaned value domains in the result relation. Similarly to the DISTINCT clause, SQL can be extended with a, say, TRIM clause that allows the user to explicitly request orphaned value domain elimination.

Second, the physical storage of tuples has to be adapted to the representation of entity domains. For tuple storage, the existing base table functionality can be reused but needs to be extended to handle uninstantiated value domains. Solutions for such an extension are manifold in the literature, e.g. interpreted record [Beckmann et al., 2006, Chu et al., 2007], vertical partitioning [Abadi et al., 2007], and pivot tables [Agrawal et al., 2001, Cunningham et al., 2004]. Another reasonable approach is a bitmap as it is used for instance by PostgreSQL [PostgreSQL Global Development Group, 2013] to mark NULL values in records. Tuples can appear in multiple entity domains. However, for storage economy and update efficiency, tuples should only appear in a single physical table. Replication should be left to explicit replication techniques. Consequently, the database system has to assign each tuple to a single physical table and maintain its logical entity domain membership. There are two principle ways how this can be done. One is to encode the domain membership in the physical table assignment. Here, the system would create a physical table for each combination of entity domains occurring in a tuple and store tuples in the corresponding table. The mapping is simple and easy to implement. On the downside, it may lead to a large number of potentially small physical tables (at worst 2^E tables where E is the number of entity domains in a database) and tuples need to be physically moved if their domain membership is changed. The other principle way is storing the domain membership, e.g. with a bitmap, directly in a tuple itself. This gives liberty regarding the assignment of tuples to physical tables, up to using a single (universal) table for all tuples. With many tuples having the same domain membership, this comes at the price of storage overhead – negligible in most cases, though.

Third, the physical tuple layout has to be extended to represent also the technical type of values directly in the tuple. This is necessary for independent technical types. To reduce storage needs and decrease interpretation overhead, the system can omit the technical type in the tuple where explicit constraints prescribe a technical type. However, creating and dropping such explicit constraints becomes expensive as the physical representation of the affected tuples has to be changed.

Fourth, independent value domains require a modification of the system catalog. In most system catalogs, value domains have a reference to the base table they belong to. This reference has to be removed to make value domains available to all tuples regardless of their entity domain membership.

2.7 Summary

With FRDM we proposed an evolutionary approach to meet the need for schema-comes-second data management and to build on the power of relational database systems. FRDM is entity-oriented instead of schema-oriented. It is designed around self-descriptive entities, where schema comes with the data and does not have to be defined up front. Additionally, FRDM allows multi-faceted entities where entities can belong to multiple entity domains. Value domains can exist independently from entity domains in FRDM. Similarly, FRDM allows to technically type values independently from their value domain. FRDM retains the flexible properties of the relational data model, which are value-based identity and value-based referencing between entities. Altogether, FRDM is a super-relational data model. It can express irregular data as well as regular relational data. We demonstrated both by examples. For data retrieval, FRDM builds on the powerful, well-known, and proven set of relational operations. Compared to the relational data model, FRDM is free of implicit constraints. Nevertheless, where these constraints are needed and welcome, the presented constraint framework FRDM-C allows formulating explicit restrictions to the flexibility of FRDM. Next to presenting FRDM and FRDM-C, we also discussed how FRDM can be implemented in existing relational database management systems. As we showed at the beginning of the chapter, FRDM has a unique set of properties, unique among other common flexible data models. FRDM combines the flexibility of graph data models with the flexibility and power of the relational model. A lot of technological expertise, knowledge, and experience have accumulated in and around relational database management systems over the last three decades. We are also convinced FRDM can contribute to the use of that experience also in the more flexibility-demanding areas of schema-comes-second data management, where traditional relational systems have been perceived as cumbersome and dated up to now.

3 ADOM – Autonomous Physical Entity Domains

The good ones go into the pot, the bad ones go into your crop.
Jacob and Wilhelm Grimm. Cinderella

With self-descriptive data, such as in FRDM, logical entity domains cannot be used as physical entity domains for storage. If they would be reused, entities belonging to multiple logical entity domains would occur redundantly in multiple physical entity domains. A straighforward solution is to centralize all entities in a single universal physical domain. Retrieval operations, however, are disadvantaged by a universal physical domain, because queries that address a single logical entity domain have to read over the whole universal domain including many irrelevant entities that do not belong to the logical entity domain in question. Horizontal partitioning presents a simple technique to increase the efficiency of such queries. A partitioning scheme taking into account the schema of self-descriptive entities allows queries to prune partitions of irrelevant entities before touching the data. Designing and maintaining such a partitioning particular for dynamic, irregular data sets is a laborious and never ending task most DBAs are not willing to commit to. Hence, online an autonomous solution is required.

Autonomous Physical Entity Domains (ADOM) is an autonomous partitioning technique for dynamic sets of entities [Herrmann et al., 2014]. It partitions an entity set into fixed-sized partitions, such that the entities in a partition share most of their entity domains. The partitions can then be used as physical entity domains. Queries that retrieve only entities belonging to a given entity domain can easily prune partitions that contain entities belonging only to irrelevant entity domains. Alongside logical entity domains, ADOM can also use other schema properties such as the instantiated value domains as a basis for the partitioning. ADOM maintains the partitioning while entities are added, modified, and removed. It is designed to keep overhead low by operating contentiously in the background; it incrementally assigns entities to partitions while they are touched anyway during modifications. This ongoing dynamic partitioning increases the locality of queries and reduces query execution cost.

In the following we define the *Online Partitioning Problem* in Section 3.1 and discuss related work in Section 3.2. In Section 3.3, we present ADOM in detail. An evaluation of the ADOM approach is presented in Section 3.4. Section 3.5 summarizes the chapter.

3.1 Online Partitioning

Entities in a universal physical entity domain are typically very heterogeneous regarding their logical entity domains and their value domains. Figure 3.1 illustrates such a universal physical domain for the product catalog scenario. As can be seen, logical entity domains as well as value domains appear irregularly among the entities. Some value domains, for instance, are very common, e.g. *name* or *weight*, while others are specific to only certain kinds of entities, e.g. *aperture* for camera with a built-in lense. The same can be seen for the logical entity domains the entities belong to. Comprehensive studies observed that distribution of logical entity domains and value domains mostly obeys Zipf's law [Chang et al., 2004, Beckmann, 2005]. Although the entities exhibit some regularity, it is hard to find reasonable and reliable generalizations that allow a static partitioning scheme. While all of today's cameras feature a sensor, a screen, and a storage card slot, some of them are also equipped with a flash, a GPS sensor, or Wi-Fi. Soon, we will see cameras with mobile connectivity but lacking a storage card slot. If there was a suitable static partitioning schema, after all, it is reasonable to assume that the database modeler would have modeled the database accordingly.

In the following, we do not distinguish between logical entity domains and value domains. From the partitioning perspective both are merely *schema properties*, which can be used as a partitioning criterion. The goal of partitioning is to increase the query efficiency by allowing the database system the early pruning of partitions with entities irrelevant for a query. Hence, it is essential that schema properties can be identified not only for entities but also for queries. Before actually executing a given query, the database system must be able to determine the set of schema properties entities must exhibit to be relevant for the query. Logical entity domains and value domains allow this.

Each partition is described in the system catalog using a partition synopsis p, which lists the schema properties of the entities in the partition. Likewise, we can list all schema properties relevant to a query in a query synopsis q. Based on the synopses, queries can easily prune partitions that contain only entities irrelevant to the query, i.e. partition for which $|p \wedge q| = 0$ holds. Correspondingly, the efficiency of a given partitioning is the ratio of how many entities are relevant to a workload and how many entities are actually read.

Definition 3.1 (Partitioning Efficiency)**.** Given a universal physical entity domain T containing the entities $T = \{e_1, e_2, \ldots\}$, a query set $W = \{q_1, q_2, \ldots\}$, and a partitioning $P = \{p_1, p_2, \ldots\}$, the efficiency of P is

$$\textsc{Efficiency}(P) = \frac{\sum_{q \in W, e \in T} \operatorname{sgn}(|e \wedge q|) \cdot \textsc{Size}(e)}{\sum_{q \in W, p \in P} \operatorname{sgn}(|p \wedge q|) \cdot \textsc{Size}(p)} \quad .$$

The function $\textsc{Size}()$ yields the size of an entity or a partition, indicating how much has to be read to scan the entity or all entities in a partition, respectively. $\operatorname{sgn}(|e \wedge q|)$

name	Device	Component	Camera	GPS	Phone	TV	Hard drive	...
Canon PowerShot S120	✓		✓					...
Sony SLT-A99	✓		✓	✓				...
Samsung Galaxy S4	✓		✓	✓	✓			...
Apple iPod touch	✓		✓		✓			...
LG 60LA7408	✓					✓		...
WD4000FYYZ		✓					✓	...
Garmin Dakota 20	✓			✓				...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

(a) Logical entity domain membership

name	resolution	aperture	screen	storage	tuner	rotation	form factor	weight	...
Canon PowerShot S120	12.1	2.0	3					198	...
Sony SLT-A99	24		3					733	...
Samsung Galaxy S4	13		4.3	32GB				133	...
Apple iPod touch	5		4	64GB				88	...
LG 60LA7408	Full HD		40		DVB-T/C/S			9800	...
WD4000FYYZ				4TB		7200	3.5"		...
Garmin Dakota 20			2.6					150	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

(b) Instantiated values domains

Figure 3.1: Example of a universal physical entity domain for electronic devices.

results in 1 if entity e is relevant to query q and 0 if not. Likewise, $\text{sgn}(|p \wedge q|)$ results in 1 if partition p contains an entity relevant for query q and 0 if not.

The aim of automatic partitioning of a universal physical entity domain is to continuously maximize the efficiency of a given partitioning under the presence of modification operations. Modification operations are inserts, updates, and deletes that change the set of entities or manipulate the schema properties of the entities.

Definition 3.2 (Online Partitioning Problem)**.** Given a universal physical entity domain T, a query set W, a partitioning P, and a modification m, online partitioning updates P so that $\textsc{Efficiency}(P)$ is maximized for W after m is applied to T.

The online partitioning problem can be solved based on the workload or solely on the entities. A workload-based solution tries to find a partitioning so that, ideally, entities in the same partition are relevant to the same set of queries. Hence, the resulting partitioning is tailored for the given workload. Whenever a workload is not available or whenever the solution should be more general and robust, an entity-based solution is more appropriate. An entity-based solution favors partitions that contain entities with attribute sets as similar as possible. The resulting partitioning is independent of a particular workload.

The online partitioning problem applies to many different database architectures and to various levels in an architecture. Most obviously in distributed databases or distributed file systems, partitions are distributed among the nodes. In modern main-memory database systems running on a large shared-memory NUMA system, partitions resemble the local memory of each CPU core. In traditional disk-based systems, pages may represent a partition granularity where solving the online partitioning problem can help to increase the query efficiency on universal tables.

3.2 Related Work

Naturally, the online partitioning problem of universal physical entity domains is related to partitioning techniques traditionally applied in database systems. Other related research fields are schema mining, inferring hidden schemas, and hypergraph partitioning. We dicuss all three in the following.

3.2.1 Traditional Partitioning

Partitioning is a well-known means of physical design. An entity set can be either partitioned vertically or horizontally. Vertical partitioning decomposes entities along attribute subsets, so that every partition contains all entities but each entity only with a disjoint subset of its values. In contrast, horizontal partitioning groups entities into partitions, so that every partition contains complete entities but only a disjoint subset of the complete entity set. There are two principal ways of horizontal partitioning. Range partitioning groups entities according to given value ranges on specified attributes; entities with values in the same value range end up in the same

partition. Hash partitioning groups entities according to a given hash function on specified attributes; entities with the same hash value end up in the same partition.

Automatic selection of a partitioning optimal for a given workload has been an active research field since the 1970s [Hoffer and Severance, 1975, March and Severance, 1977]. Partitioning advisor tools became a popular research topic along with the research on index advisor tools [Rao et al., 2002, Agrawal et al., 2004, Zilio et al., 2004]. An online partitioning tool was presented only recently [Jindal and Dittrich, 2011]. For horizontal partitioning, all these tools consider mainly range partitioning because it is the partitioning mostly used in traditional relational database system setups. In web-scale databases, where load balancing over a large number of node is the main concern, hash partitioning is the common choice [Chang et al., 2006, DeCandia et al., 2007, Lakshman and Malik, 2010]. To the best of our knowledge, horizontal partitioning based on schema properties has not been considered so far.

3.2.2 Schema Mining

Schema mining aims at finding interesting structures and structural regularities in databases of semi-structured data. The findings can be used to learn about unknown data or to describe and type irregular data in a compact way. Schema mining became a popular research topic in the second half of the 1990s, when the rise of the Web rapidly increased the amounts of irregular, semi-structured data. Commonly, schema mining approaches use an edge-labeled graph as in the object exchange model [Papakonstantinou et al., 1995] to represent semi-structured data. They differ, however, in the technique used for the actual mining.

Nestorov et al. [1998] use clustering. In the first stage of their approach they identify the minimal set of structural types that covers the complete data set using greatest fixpoint semantics from logical programming. For fairly irregular data, this minimal set of structural types still encompasses a large number of types. In the second step, the types are clustered into k groups of similar types relying on the L_1 distance. Per group, they coalesce the types into a single type, which represents the cluster in the schema mining result.

Wang and Liu [1998, 2000], in contrast, extract frequent structures using the notion of minimum relative support from frequent itemset mining [Agrawal et al., 1993, Agrawal and Srikant, 1994]. Given a set of data objects and a minimum relative support of x percent, they identify all structures that occur in at least x percent of the data objects. Basically, Wang's algorithm generates candidate structures and checks their support; pruning techniques help reduce the candidate set. Nevertheless, the set of frequent structures likely contains many redundancies, because for a larger frequent structure all its substructures are frequent, too. They call these substructures weaker since their typing information is less comprehensive. A maximally frequent structure is frequent and not weaker than other frequent structures. By checking this condition, Wang and Liu simply filter out all nonmaximally frequent structures. The approach of Laur et al. follows the same lines, though they claim to achieve better results [Laur et al., 2000].

Although schema mining is a very closely related topic, the schema mining techniques are not applicable to our horizontal partition problem. While schema mining finds frequent structures in a data set, it does not unambiguously decide which data objects belong to which frequent structure. With all of the discussed techniques, a data object can exhibit multiple frequent structures. Schema mining does not partition the data; it only finds representative structures.

3.2.3 Inferring Hidden Schemas

Chu et al. [2007] consider a problem very closely related to schema mining as well as our online partitioning problem. In their approach, they also partition a universal physical entity domain but vertically and offline. Although they called their approach Inferring Hidden Schemas, the overall goal is identical to ours. Chu et al. use the discovered hidden schema of a universal physical entity domain to increase retrieval efficiency by reducing the data that has to be read, and to minimize the irregularity per partition. Their partitioning algorithm clusters attributes based on their co-occurrence. They measure the co-occurrence of two attributes a and b with the Jaccard coefficient of the two sets A and B, where A is the set of tuples instantiating attribute a and B is the set of tuples instantiating attribute b. Accordingly, the coefficient of a and b measures as $|A \cap B| \,/\, |A \cup B|$. In the extreme, the coefficient equals zero if none of the entities instantiate both attributes, or one if a and b only occur together. With the coefficients Chu et al. create an adjacency matrix and apply a k-nearest-neighbor clustering to obtain the partitions.

The quality of the resulting partition depends on the parameter k of the clustering algorithm. Unfortunately, the authors do not explain how to obtain a good k for a given data set. Without any additional knowledge about the data set k is practically unknown. Plus, the good k may change over time while the data set evolves. For evaluation purposes, the authors provide a measure for the quality of a partition, which quantifies the sparseness in a partition. We could use this measure to find a good k in two ways. First, we can compare the quality of partitionings resulting from different settings of k. This solves the problem, but significantly increases the computational complexity as we have to run the clustering algorithm for every reasonable setting of k. Second, we can implement a search strategy on k and stop if we have found a reasonably good partitioning by setting a threshold on the quality measure. This, however, leaves us with the same kind of question: What is a good threshold? Like a good k, a good threshold is practically unknown if we do not have any additional knowledge about the data set.

The hidden schemas approach works offline and partitions vertically. Nevertheless it can generally be applied to our online horizontal partitioning problem. By maintaining the adjacency matrix incrementally and reruning the k-nearest-neighbor clustering after any change we can turn the concept into an online algorithm. However, the online version will come at considerable overhead, since the clustering algorithm always runs on the complete matrix although only a small fraction of the coefficients has changed. The clustering will basically make the same global decision over and

over again. The other downside of the concept is that it partitions vertically. Vertical partitions decompose entities. Depending on the irregularity of a data set, an entity may occur in multiple partitions, which requires joins on retrieval to reconstruct the entities. For the same reason, data manipulation operations may have to address multiple partitions, as the authors freely admit.

We can conclude that the hidden schema concept is closely related to our problem. Although the algorithm is applicable to our problem, it does not represent a reasonable solution. It requires a hard-to-set parameter and comes with considerable overhead for the partitioning itself and for query processing.

3.2.4 Hypergraph Partitioning

Graph partitioning divides the vertex set of a graph into disjoint subsets. A good graph partitioning minimizes the number of edges that connect vertices from different partitions. Hypergraph partitioning is common extension of graph partitioning. Hypergraphs allow edges to connect more than two vertices. Graph partitioning in general and hypergraph partitioning specifically are NP-hard problems. Many heuristics and approximation algorithms have been developed in various application areas over the years. Hypergraph partitioning is used for VLSI [Leighton et al., 1990, Karypis et al., 1997, 1999], for sparse matrix decomposition in parallel computing [Çatalyürek and Aykanat, 1996, 1999], and for declustering large databases to parallel disks [Fang et al., 1986, Liu and Shekhar, 1996].

Hypergraph partitioning is also applicable to our partitioning problem. Given an instance of the partitioning problem, we can construct a hypergraph such that the partitioning of the vertices using the *connectivity* criterion yields the optimal solution for that instance.

Figure 3.2 illustrates the optimal solution by a simple example, consisting of the entities $e_1, \ldots, e_6$ and the queries $a, \ldots, f$. For each query only a subset of entities is relevant, e.g. for query e only entity e_4 and e_6 are relevant. In the hypergraph, each vertex represents an entity and each hyperedge represents a query, such that the edge connects exactly those vertices that are relevant for the query. For the partitioning, the connectivity criterion minimizes the total number of partitions all hyperedges connect. In the context of entity partitioning, the connectivity corresponds to the number of distinct partitions that can contain relevant entities for a query. In the example of Figure 3.2, the optimal partitioning of the entities $e_1, \ldots, e_6$ for given queries is $\{\{e_1, e_2, e_3\}, \{e_4, e_5, e_6\}\}$. The other partitioning shown in the figure has a high connectivity and therefore is not an optimal solution.

The construction of the graph is based on a specific workload, hence, the result of the partitioning will be tailored for this workload. We call it the *workload-based* solution to the problem. Whenever a workload is not available, or where the solution should be more general and robust, we can base the hypergraph on the schema properties of the entities. The procedure is similar to the one shown in Figure 3.2, except that $a, \ldots, f$ are schema properties, i.e. entity 4 and 6 feature schema properties e. We call

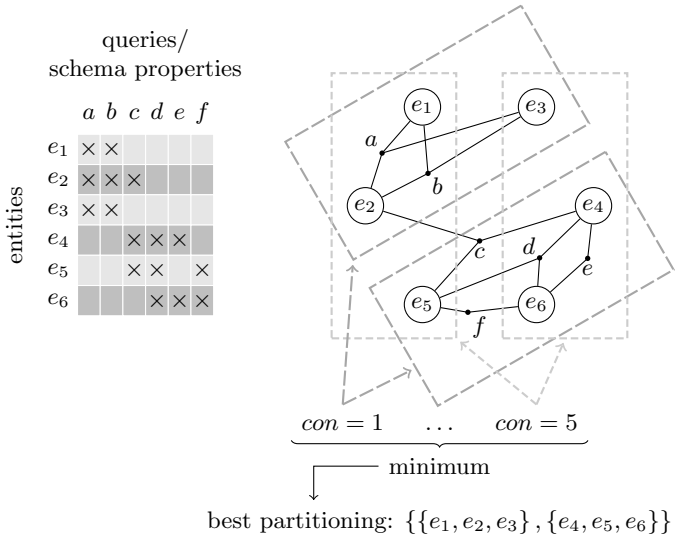


Figure 3.2: Optimal Solution using hypergraph partitioning.

the partitioning scheme resulting from such a hypergraph the *schema-properties-based* solution to the problem.

To solve the online partitioning problem, workload-based or schema-properties-based, we could create the hypergraph for all entities including the one affected by the modification, and determine the partitioning. Unfortunately, this is unfeasible in practical scenarios as hypergraph partitioning is NP-hard [Garey and Johnson, 1979]. Even with heuristic hypergraph partitioning algorithms, the solution comes with considerable overhead for two reasons. First, the approach represents each individual entity in the hypergraph, which causes an enormous amount of data accesses. Second, it recalculates the complete partitioning with each modification operation, although most of the entities remain unchanged. A practical solution to the online partitioning problem has to exhibit small overhead, such that the overhead does not exceed the benefit achieved by the partitioning scheme.

3.3 ADOM

ADOM is specifically designed for the Online Partitioning Problem. It consists of two main components: (1) the partitioning maintenance algorithm and (2) the partition rating. We detail both in the following.

3.3.1 Partitioning Maintenance

ADOM works incrementally. It relies on the basic assumption that the data is already well partitioned. Triggered by a modification operation, ADOM merely adjusts the partitioning so that the modified entity fits in well. ADOM creates partitions of a

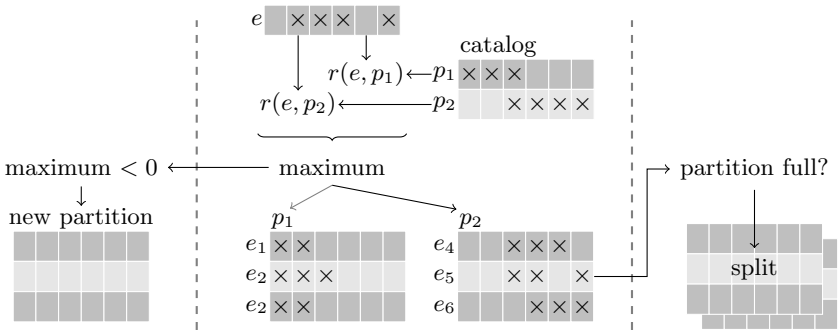


Figure 3.3: Insert procedure.

fixed maximum size. Partitions that reach their capacity limit are reorganized with a split operation. Since, among the common data modification operations, inserts affect the partitioning most, we will focus the discussion of ADOM on the insert operation. ADOM can create a workload-based as well as an schema-properties-based solution. For simplicity in the discussion, we will assume the schema-properties-based setup.

The basic insert procedure is illustrated in Figure 3.3. Given two partitions cataloged with their partition synopses and a new entity, ADOM scans the partition catalog to find the partition which fits best to the entity. Every partition is rated and the entity is inserted to the partition with the highest rating. We will discuss the rating in detail in Section 3.3.2. There are two possible exceptions from this basic procedure, illustrated on the left and the right of Figure 3.3. First, the rating can become negative, indicating that the new entity fits none of the existing partitions well. In this case, ADOM creates a new partition for the new entity. Second, the highest rated partition has reached the maximum capacity B. Here, ADOM splits the partition into two new partitions.

For the split the insert procedure maintains a pair of so-called split starters for each partition in the system catalog. The split starters are two entities from a given partition that differ as much as possible in their synopses. The first two entities added to a partition form the initial pair of split starters. With every additional entity added, the insert procedure checks whether this entity would make a better, i.e. more differential, starters pair with one of the original starter entities. If that is the case the insert procedure updates the partition's pair of split starters accordingly. The difference between entity synopses e_1 and e_2 is calculated as the number of different schema properties $|e_1 \oplus e_2|$. This incremental maintenance heuristically tries to maximize the difference of the split starters. It does not guarantee to yield the most differential pair of entities in a partition, but it avoids the cubic effort necessary to determine the most differential pair.

To split a partition, the insert procedure creates two new partitions and moves each of the two split starter entities to one of the new partitions. The remaining entities are assigned to the new partitions using the insert procedure itself, while limiting the set of possible target partitions to the two new partitions. The procedure does

not necessarily result in a balanced split; the result strictly depends on the schema properties of the involved entities. The recursive use of the insert procedure can result in a split cascade. Neither will such a split cascade be very long nor is it a very likely event.

Algorithm 1 lists the complete insert routine. First, ADOM iterates over the partition catalog to find the partition rated best for the entity to insert (lines 4–8). Then, depending on the best rating, the algorithm either creates a new partition for the entity (lines 10–14), splits the best rated partition (lines 27–34), or simply inserts the entity into the best rated partition (line 37). In the case of a split or a normal insert, ADOM updates the pair of split starters to reflect the new entity (lines 16–25). Specifically, the new entity will replace one of the split starters if it has a difference to the other split starter larger than the difference between the original split starters.

As can be seen, the complexity of finding the best rated partition depends on the number of partitions and the cardinality of the synopses. For a split, the complexity depends on the partition size and, again, the cardinality of the synopses. However, in I/O bound systems the runtime will be dominated by the moving of the actual entities from partition to partition. Consequently, the split, although linear in complexity, is the most expensive part of the algorithm.

The adjustment routines that ADOM performs for the other modification operations rely on the insert routine. Upon deletes, ADOM merely removes the deleted entity from its partition. The partitioning itself remains unchanged. Empty partitions will be deleted. Upon updates, ADOM also runs the insert routine but without actually inserting. In case the updated entity is assigned to a new partition it is moved. Otherwise, ADOM updates the entity in place.

Assigning entities to partitions during insert and split relies on the rating of how well an entity and a partition fit together. The next section explains this rating in more detail.

3.3.2 Partition Rating

ADOM's partition rating compares an entity synopsis with a partition synopsis to determine how well the entity would fit in the partition. We first discuss the local rating, which is not comparable among partitions, and then present how ADOM obtains a comparable global rating from the local rating.

Figure 3.4 illustrates the idea of the local rating r'. The rating takes positive evidence as well as negative evidence into account. Positive evidence is the amount of regularly structured data a partition will contain if the entity is added. We refer to this postive evidence also as homogeneity between entity and partition. ADOM determines the evidence as the number of attributes that can be found in both entity and partition, multiplied with the sum of the size of the partition and the size of the entity.

Homogeneity score: $\text{homo}(e, p) = (\textsc{Size}(p) + \textsc{Size}(e)) \cdot |e \wedge p|$

Algorithm 1 Online horizontal partitioning.

```
1:  procedure InsertEntity(e, s_e, C)                          ▷ e: entity
2:                                                             ▷ s_e: entity synopsis
3:                                                             ▷ C: catalog with partition synopses
4:      r_best ← −∞
5:      for (s_p, p) ∈ C do                                    ▷ scan partition synopses in catalog
6:          r ← Rate(s_e, s_p)                                 ▷ calculate rating
7:          if r_best < r then                                 ▷ if rating is the best so far
8:              r_best ← r;   p_best ← p                       ▷ save current best
9:      ▷ if best rating is negative
10:     if r_best < 0 then
11:         p_best ← CreateNewPartition()
12:         p_best.Add(e)                                      ▷ add entity to new partition
13:         p_best.e_A ← e                                     ▷ set entity as first split starter
14:         return
15:     ▷ update split starters of partition r_best
16:     if p_best.e_B = Null then                              ▷ if second split starter is missing
17:         p_best.e_B ← e                                     ▷ set entity as second split starter
18:     else                                                   ▷ check if new entity is a better split starter
19:         r_eA ← Diff(e, p_best.e_A)
20:         r_eB ← Diff(e, p_best.e_B)
21:         r_AB ← Diff(p_best.e_A, p_best.e_B)
22:         if r_eA = Max(r_eA, r_eB, r_AB) then
23:             p_best.e_B ← e                                 ▷ e becomes split starter B
24:         else if r_eB = Max(r_eA, r_eB, r_AB) then
25:             p_best.e_A ← e                                 ▷ e becomes split starter A
26:     ▷ if partition is full, split
27:     if Size(p_best) + Size(e) > MaxSize then
28:         p_A ← CreateNewPartition()
29:         p_B ← CreateNewPartition()
30:         p_A.Add(p_best.e_A);   p_best.Remove(p_best.e_A)
31:         p_B.Add(p_best.e_B);   p_best.Remove(p_best.e_B)
32:         for e_split ∈ p_best do                            ▷ split remaining entities
33:             InsertEntity(e_split, s_e_split, {p_A, p_B})
34:             p_best.Remove(e_split)
35:         return
36:     ▷ normal case: just add entity to partition r_best
37:     p_best.Add(e)
```

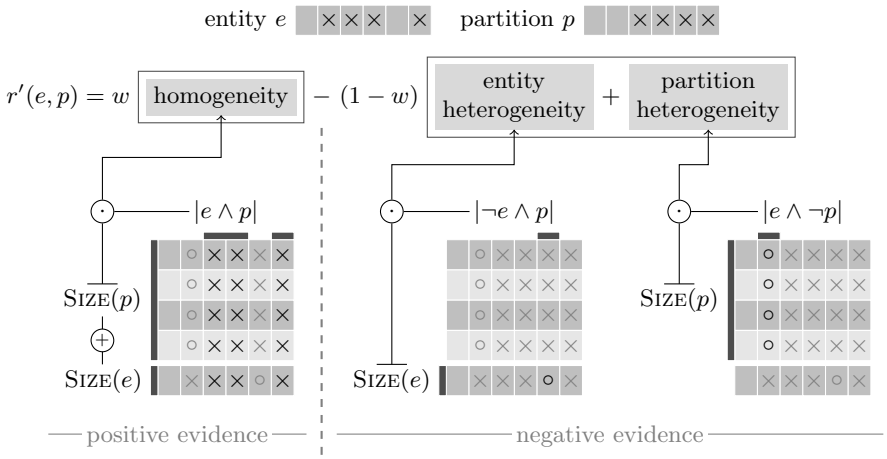


Figure 3.4: ADOM's local rating.

Negative evidence is the amount of irregularly structured data that will result from adding the entity to a partition – in Figure 3.4 indicated by the black circles. There are two kinds of negative evidence. The first one is heterogeneity on the part of the entity and originates from attributes the partition has but the entity lacks. Analogously to the homogeniety, we measure this entity heterogeneity as the number of the entity's missing attributes multiplied by the size of the entity.

Entity heterogeneity score: $\text{hetero}_e(e,p) = \textsc{Size}(e) \cdot |\neg e \wedge p|$

The second negative evidence is heterogeneity on the part of the partition and originates from attributes the entity has but the parition lacks. We measure this partition heterogeneity as the number of the partition's missing attributes multiplied by the size of the partition.

Partition heterogeneity score: $\text{hetero}_p(e,p) = \textsc{Size}(p) \cdot |e \wedge \neg p|$

Note that ADOM heuristically assumes that the partitions it creates are rather homogeneous. Irregularity already existing in a partition is not considered since this would require more information than the simple synopses.

To get a local rating r', ADOM subtracts the total of the negative evidence from the positive evidence.

$$r'(e,p) = w \cdot \text{homo}(e,p) - (1-w) \cdot (\text{hetero}_e(e,p) + \text{hetero}_p(e,p))$$

The weight w allows the DBA to balance between the influence of positive and negative evidence. For a given dataset, weights greater than 0.5 result in a smaller number of more heterogeneous partitions, while weights less than 0.5 result in a larger number of rather homogeneous partitions. In the extreme setting of $w = 0$, any heterogeneity will result in a negative rating. As a result, ADOM creates only perfectly homogeneous partitions like in a normal database for regularly structured data. The other extreme

setting of $w = 1$ results in partitions similarly heterogeneous as the original universal entity domain. Our experimental results suggest that a weight between 0.3 and 0.6 is a reasonable setting.

The local rating r' is not comparable between partitions because the amount of data and size of the attribute set varies from partition to partition. To compensate for this, ADOM uses the global rating r which is normalized with the partition size and the number of the involved attributes:

$$r(e,p) = \frac{r'(e,p)}{(\textsc{Size}(p) + \textsc{Size}(e)) \cdot |e \vee p|} \quad .$$

3.4 Evaluation

We created a prototypical implementation of ADOM and studied ADOM's performance on the irregularly structured data of DBpedia. Additionally, we used the TPC-H benchmark to evaluate ADOM on regularly structured data. In this section, we first discuss the setup, which contains details on the used implementation, and then present the results.

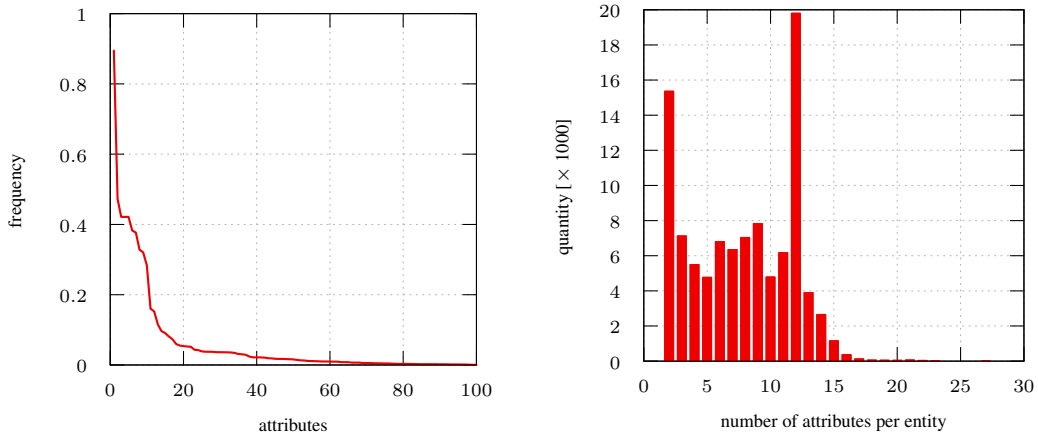
3.4.1 Setup

We implemented ADOM in PostgreSQL 9.3 using views, triggers, and stored procedures. In our implementation, a universal table simulates a universal physical entity domain. Each data manipulation operation on the universal table triggers a stored procedure, which implements the ADOM algorithms. The prototype creates a regular table for each partition as well as a single catalog table for the metadata of all partitions. ADOM uses the metadata to rewrite incoming queries to a `UNION ALL` over all relevant partitions. The prototype provides transparent data access through ADOM, as the user inserts data to the universal table using regular SQL statements.

Hardware platform for our experiments was a Windows 8 computer with an i7 CPU and 8 GB of memory. We executed two kinds of experiments. With data taken from DBpedia, we evaluated how much selective queries on irregularly structured data can benefit from ADOM. We also studied the influence of ADOM's two main parameters, the partition size limit B and the weight w in the DBpedia setup. With the TPC-H benchmark, we investigated the effect of ADOM on regularly structured data. In the following, we detail the two experiments on irregularly and regularly structured data.

3.4.2 Irregularly Structured Data

DBpedia [Auer et al., 2007] is a large database of irregularly structured data, created by a diversity of web users. Given DBpedia's heterogeneity, selective queries can benefit from horizontal partitioning by early elimination of irrelevant partitions from data access. For this experiment, we extracted 100 000 person entities with a total of 100 attributes. Although we limit the data set to person records, it exhibits the typical long tail distribution of irregularly structured data. Figure 3.5 shows the distribution



(a) Attribute frequency.

(b) Number of attributes per entity.

Figure 3.5: Attribute distribution in the DBpedia data set.

of the attribute in the data set. In particular, Figure 3.5(a) shows the distribution of the attribute frequency, i.e. how many entities of the data set instantiate a given attribute. As can be seen, two attributes are extremely common and appear on almost every entity. Eleven attributes are fairly common and appear on over 30 % of the entities, while 85 % of the attributes appear on less than 10 % of the entities. Figure 3.5(b) shows the distribution of the number of attributes per entity. While the majority of entities have between two and 15 attributes, a few entities have up to 30 attributes.

We inserted the data into an ADOM-partitioned universal table, under different settings of partition size limit and weight. The DBpedia person entities were inserted in random order. In the process, we measured the execution of the inserts; afterwards we recorded metrics about the partitioning ADOM had created in the particular setting and measured the execution time of selective queries. For this purpose, we generated a synthetic workload since there is no common or standardized DBpedia workload. The goal was to obtain queries with different selectivity to evaluate the effect of ADOM according to the query selectivity. We created multiple sets of attributes. Each of the individual attributes forms an attribute set. Additionally, we combined the 20 most frequent attributes to pairs and triples. For each of these attribute sets we generated a query of the form

```
SELECT a1, a2, ...  FROM universalTable
WHERE a1 IS NOT NULL OR a2 IS NOT NULL ...
```

where $\{a_1, a_2, \ldots\}$ is the corresponding attribute set. Each of these queries returns only entities that instantiate the given attributes. The selectivity of the queries varies depending on the attributes queried. We collected representative queries to reasonably cover the range of possible selectivities; three representative queries for each selectivity.

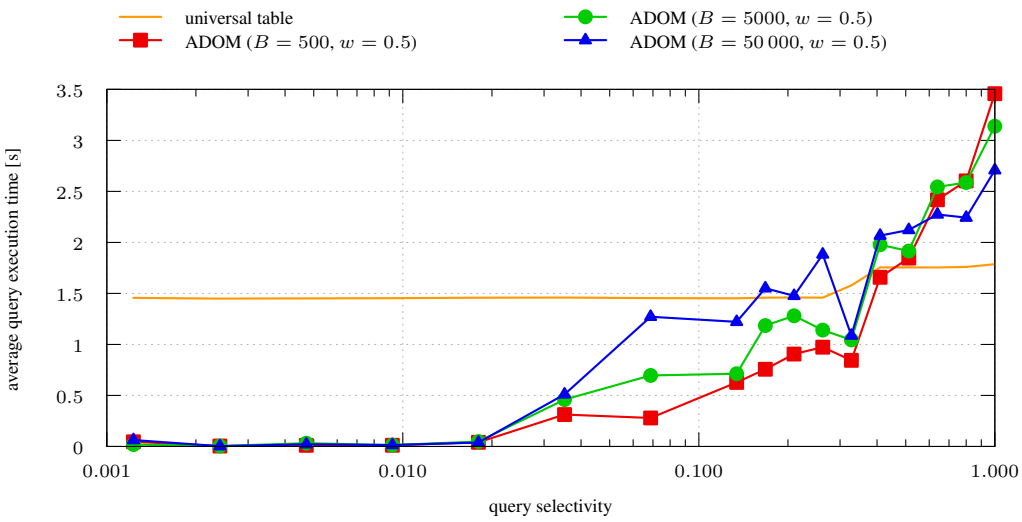


Figure 3.6: Average query execution time for different partition size limits B.

We start the discussion of the results with the measurements of query execution time. This is followed by a more detailed study of the influence of the weight in the partition rating on the resulting partitioning. Finally, we discuss the measurements of insert execution time.

Query Execution Time

For the representative queries of our synthetic workload, we measured the execution time using different partition size limits and, for comparison, the execution time on the original universal table. Neither the ADOM partitions nor the universal tables had an index on any column. Figure 3.6 shows the average query execution times depending on the selectivity for a partition size limit of 500, 5000, and 50 000 entities. The weight was set to 0.5. As expected, the query execution time increases with a decreasing selectivity of the queries since more data has to be read. In contrast, the query execution time increases only slightly on the original universal table. Regardless of the actual selectivity, queries have to scan the whole table here. Consequently, ADOM achieves a significant speedup for very selective queries (selectivity < 0.04). In general, the more entities a query reads the smaller the benefit of horizontal partitioning is. For queries of medium selectivity (> 0.04 and < 0.4), ADOM achieves lower or about the same query execution time as with the original universal table. Queries of low selectivity (> 0.4) are likely to access every partition and do not profit from ADOM. With our prototype, these queries show a longer execution time with ADOM than on the universal table. We attribute large parts of this overhead to the implementation of our prototype. For instance, during the union operation, the database system has to project all tuples of every involved partition to the common schema. This cost can be avoided by an implementation directly in a FRDM database

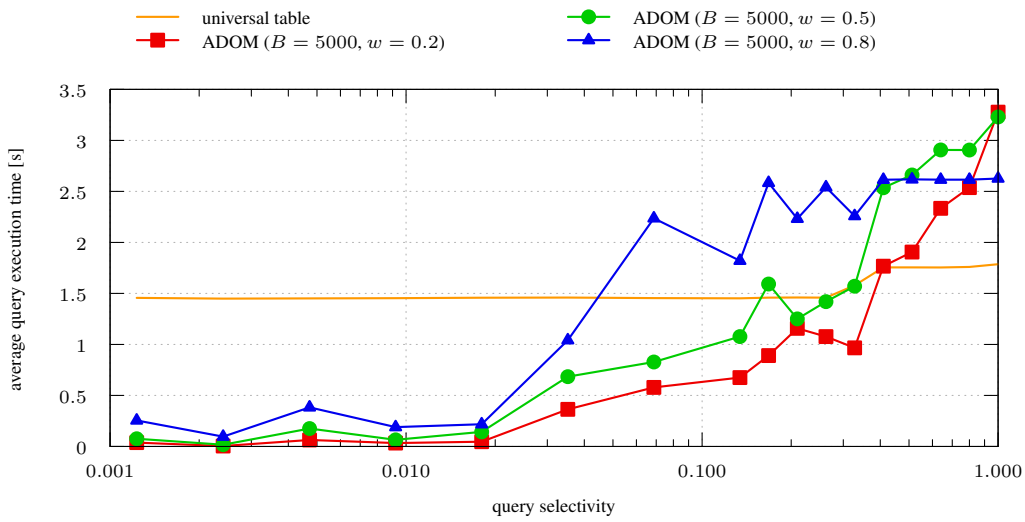


Figure 3.7: Average query execution time for different weights w.

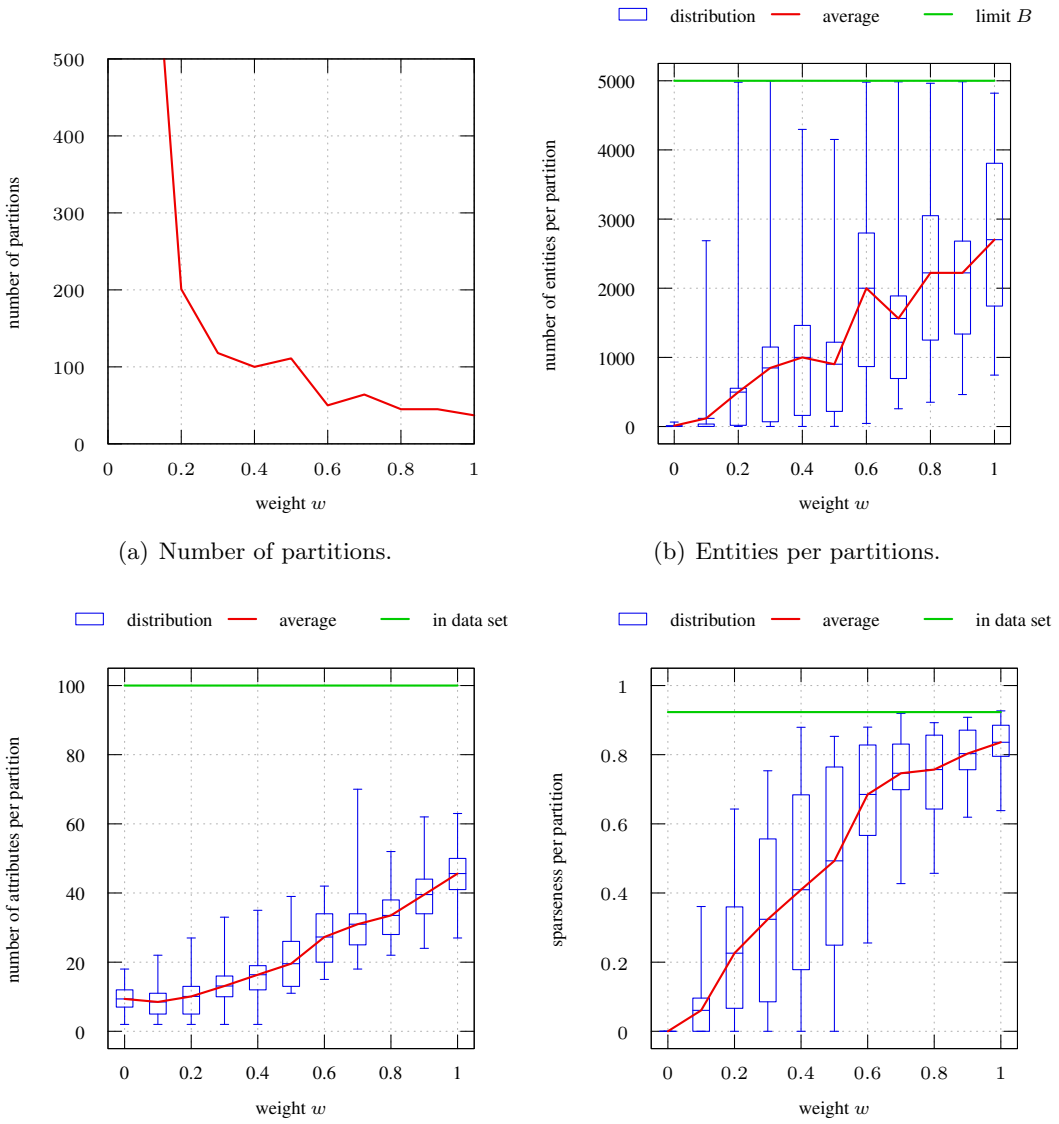
engine with a native presentation for irregular data. There, we expect low selectivity queries not to suffer more than queries suffer from normal range or hash partitioned tables. Nevertheless, the aim of ADOM is to speed up selective queries and it is seen to be capable of doing so.

Figure 3.6 also shows the impact of the partition size limit on ADOM's benefit. Given a data set, a smaller partition size limit allows ADOM to build more homogeneous partitions. This leads to a lower query execution time, particularly for queries of medium selectivity. On the downside, a smaller partition size limit also results in a larger number of partitions necessary to host the data. This requires less selective queries to unite more partitions, which increases the overhead for such queries. Consequently, the partition size limit should be set lower for very selective workloads and higher for less selective workloads.

Figure 3.7 shows the impact of the weight of the partition rating on ADOM's benefit. We see a similar picture here. Higher weights typically result in fewer but larger partitions. For very selective queries, a lower weight is benefical, while queries of very low selectivity profit from a higher weight. However, the optimal weight depends more on the irregularity of data set than on the workload. For the DBpedia data set we used in the evaluation, 0.2 is seen to be a good balance between positive and negative evidence in the partition rating. For other data sets with another irregularity another weight is likely to be optimal.

Influence of Weight on Partitioning

For a more detailed picture of the influence of the weight w on ADOM's partitioning, we partitioned the DBpedia data set with ADOM using different settings for the weight. For each of the resulting partitionings, we recorded (1) the number of partitions, (2)



(a) Number of partitions.

(b) Entities per partitions.

(c) Attributes per partitions.

(d) Sparseness per partition.

Figure 3.8: Influence of the weight w on the partitioning of the DBpedia data set.

the number of entities per partition, (3) the number of attributes per partition, and (4) the sparseness per partition. The maximum partition size limit B was set to 5000. Figure 3.8 illustrates the results.

As Figure 3.8(a) clearly shows, the lower the weight the more partitions ADOM creates. Particularly with a weight less than 0.2 the number of partitions explodes. With a very low weight, the homogeneity score loses its influence on the partition rating. As a result, most entity–partition pairs get a negative rating, causing ADOM to create a new partition even for entities that have a large overlap with the schema of existing partitions. In the extreme case of $w = 0$ all created partitions are completely homogeneous.

Naturally, the number of entities in the partitions behave the opposite way, as shown in Figure 3.8(b). Higher weights allow more heterogeneity within partitions, so that more entities are assigned to the partitions. With a very low weight ($w < 0.2$), all partitions remain very small. Medium weights produce a few partitions filled to the maximum capacity, although the majority is less than half full. The large spread results from the attribute distribution. On the one hand, a large fraction of entities has only a small number of attributes and many of these attributes are also the most common attributes. Hence, these entities pile up in a large partition. On the other hand, a very small fraction of entities has a large number of very uncommon attributes. These entities result in very small partitions.

Figure 3.8(c) shows the number of attributes per partition. Again, the higher the weight the more heterogeneity per partition is allowed and the higher the number of attributes. However, in all settings all partitions have significantly fewer attributes than the universal table has. This clearly shows how ADOM facilitates the pruning of irrelevant data. The lower the number of attributes the higher the probability that a partition can be pruned from query processing.

Since the number of entities and the number of attributes per partition increase with the weight, the sparseness per partition has to increase as well for a given data set. Figure 3.8(d) shows this effect for the DBpedia data set. For the homogeneity-preserving setting of $w = 0$, the sparseness per partition is obviously zero. In contrast, higher weights ($w > 0.6$) do not form any partition of very low sparseness. With all medium weight settings, ADOM forms mainly partitions with a sparseness considerably lower than the sparseness of the original data set.

All results in Figure 3.8 underline that medium weights produce the most reasonable results. A perfect partitioning has to reach contradicting goals. On the one hand a small number of partitions that are well filled up to the partition size limit minimizes the overhead for queries of low selectivity. On the other hand very dense partitions with small attribute sets facilitate the highest speedup for very selective queries. Medium weights balance between these contradicting goals.

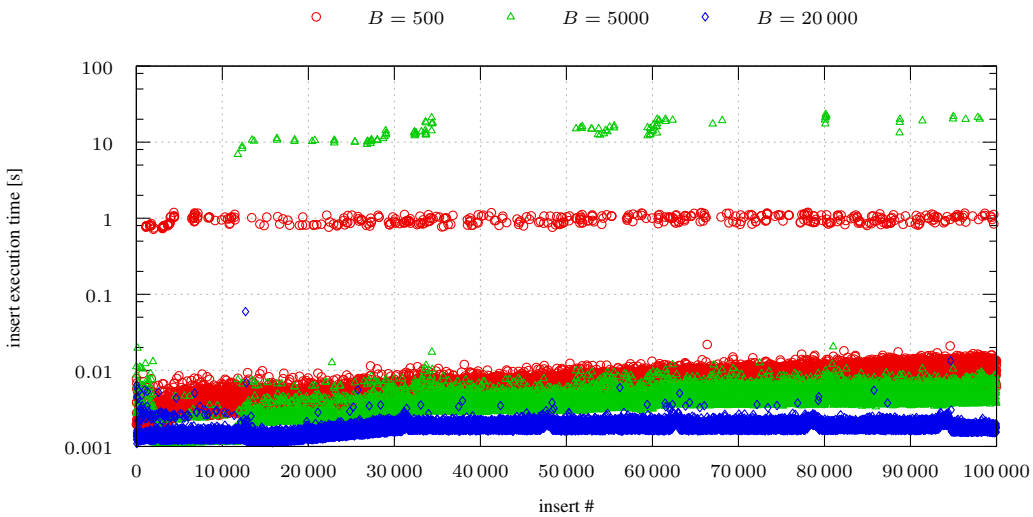


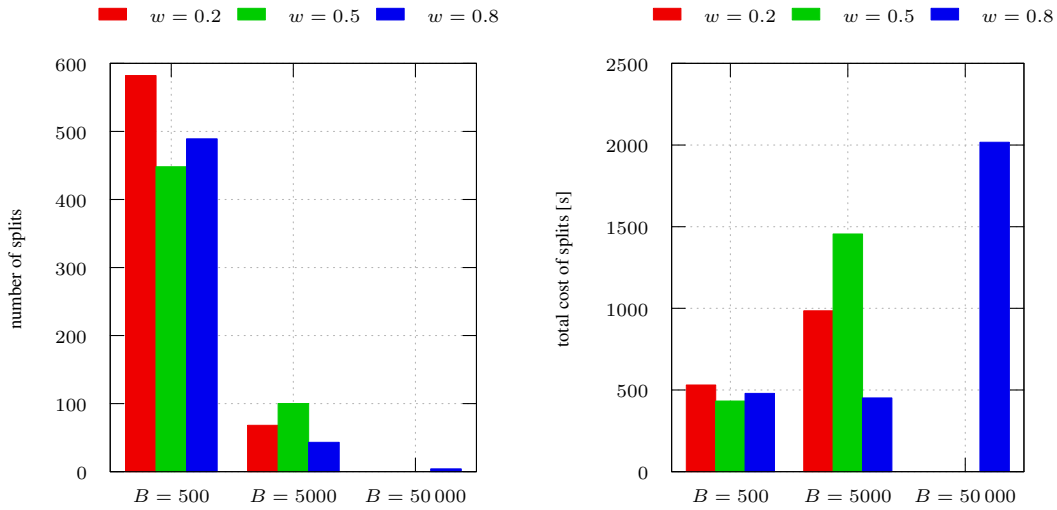
Figure 3.9: Insert execution time for different partition size limits B.

Insert Execution Time

We also measured the execution time of the insert operations, when we loaded the data set. Figure 3.9 shows the results[1] for different partition size limits and a weight of 0.5. As can be seen, the majority of insert operations finishes in between 1 ms and 10 ms. With a lower partition size limit and a larger number of partitions, the inserts take a little longer, because the size of the partition catalog increases. A small fraction of inserts needs considerably longer, though. These are insert operations where a partition split occurs. As Figure 3.9 clearly shows, the number of insert operations with a split decreases with an increasing partition size limit. With a partition size limit of 500 entities, ADOM performs a split 448 times, for a limit of 5000 entities 100 times, and for a limit of 50 000 no split occurs. At the same time, the cost of a split increases with the partition size limit, because more entities have to be reassigned and physically moved during a split.

Figure 3.10(a) shows the number of splits occurring with the DBpedia dataset depending on partition size limit and weight. While there is no clear tendency for the influence of the weight, the partition size limit is seen to be the main influence on the number of splits. As we have seen before, the lower the partition size limit the more splits. The total cost of the splits forms a different picture, as depicted in Figure 3.10(b). Fewer splits do not necessarily translate into less overhead. Since the cost of an individual split increases drastically with the partition size limit, the total cost of all splits is also higher for larger partition size limits. Note that lazy

[1] For values below 0.4 s, the figure shows only a 10 % sample. For every 50 inserts of the same setting the sample includes five values – the maximum, the minimum, and three other randomly picked values.



(a) Number of splits.

(b) Total cost of splits.

Figure 3.10: Influence of partition size B and weight w on splits.

maintenance strategies known from materialized views [Zhou et al., 2007] can help to ease the effect that a high split cost has on a productive database system.

3.4.3 Regularly Structured Data

The TPC-H benchmark [Transaction Processing Performance Council, 2013] has, in contrast to DBpedia, a concrete specification of the workload. For this experiment, we loaded TPC-H data with a scale factor of 0.5 into an ADOM-partitioned universal table. Views on the partitions created by ADOM emulated the standard TPC-H tables. The TPC-H data is perfectly regular. While inserting this data into an ADOM-partitioned universal table, ADOM should be able to find a partitioning which is similar to the TPC-H table schema. Any partitioning different from the TPC-H schema would result in a significantly higher execution time for the TPC-H queries. Accordingly, we measured the execution time of the benchmark's queries on the TPC-H-schema-emulating views using different partition sizes and, for comparison, the execution time on the regular tables of the TPC-H schema.

Table 3.1 shows the results. The table lists the total execution time of the 22 TPC-H queries in four scenarios. One scenario is the normal TPC-H benchmark without ADOM and provides the baseline. The other three scenarios show ADOM with different settings for the partition size. As can clearly be seen, the presence of ADOM adds only a small overhead to the query execution time on regularly structured data. In fact, ADOM finds only partitions which exactly fit the TPC-H schema in any of the three settings. Again, we see that a larger partition size decreases the cost of the additionally necessary union operations for queries that span multiple partitions.

Table 3.1: Query execution time on regular data (TPC-H).

Scenario	Partition size limit	Total query execution time	
Standard TPC-H	–	24.23 s	(100.00 %)
ADOM I	500 entities	26.38 s	(108.87 %)
ADOM II	2000 entities	25.61 s	(105.69 %)
ADOM III	10 000 entities	24.54 s	(101.27 %)

3.5 Summary

Universal physical entity domains are a common setup in databases involving a significant share of irregular, self-descriptive entities. Horizontal partitioning can help to increase the efficiency of queries on such universal physical entity domains. Maintaining such partitioning poses an optimization problem in the field of physical design. We defined this as the Online Partitioning Problem for heterogeneous data. With ADOM, we proposed an autonomous online algorithm for horizontal partitioning of self-descriptive entities. ADOM separates entities into partitions of fixed maximum size, which can be used as physical entity domains in a database management system. ADOM creates the partitioning based on the schema properties of the entities, which allows queries to prune partitions of irrelevant entities from processing before actually touching them. In the evaluation, ADOM was seen to be capable of significantly increasing the query efficiency over a universal physical entity domain for selective queries. We therefore consider ADOM a good autonomous solution to automatically determine physical entity domains for FRDM data.

4 FASE – A Freely Adjustable Storage Engine

> Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation).
>
> *Edgar Frank Codd [Codd, 1970]*

Irregular, self-descriptive data with variable schema requires an alternative physical data organization. The physical data layout has to provide flexibility for frequent schema changes and for highly irregularly structured data. Several physical data layouts suitable for irregular data have been proposed in the literature [Boncz and Kersten, 1999, Agrawal et al., 2001, Cunningham et al., 2004, Beckmann et al., 2006, Chu et al., 2007, Ooi et al., 2007, Abadi et al., 2007, Weissman and Bobrowski, 2009]. All of them have their advantages and disadvantages. The physical data layout that is optimal for a given database very much depends on the workload and the actual irregularity of the data. Data irregularity, however, is a variable characteristic of data. Different data sets typically show different degrees of data irregularity, depending on the application domain, the original data source, and how the data is integrated or inserted into the database. Something similar holds true within a single data set. Typically, certain parts of the data exhibit high irregularity while others are regularly structured. In database systems where data is integrated and consolidated, irregularity can also vary over time. Consequently, there is no single apriori physical data layout that is optimal for schema-comes-second databases.

FASE is a storage engine that is configurable in respect of the physical data layout. It does not implement a single physical data layout, but can be set up to a number of different layouts. FASE builds on a declarative storage description language for the macroscopic characteristics of the physical data layout – the FASE notation. The FASE notation allows expressing of the grouping and clustering of data. Once configured accordingly, FASE shows the same fundamental performance tradeoffs of the different physical data layouts as hard-coded implementations – all within a single system. In the evaluation, we show how the performance of FASE varies depending on the configured physical data layout and how different types of queries benefit from certain physical data layouts.

FASE does not aim to compete with hard-coded data layouts for performance but strives for generality. Although achieving top-level performance, specialized database management systems increase the complexity of every IT landscape. With the additional complexity and cost they bring, specialized systems are not affordable to every extent and for every customer. They are only worthwhile where absolute top performance is needed. In most cases, a single system with a configurable physical

(virtual) id	a	b	c
1	1	1	1
2	1	1	4
3	1	2	4
4	1	2	5
5	1	2	5
6	2	1	1
7	2	1	1
8	2	3	1
9	2	3	2
10	2	3	2
11	2	3	3
12	2	3	4

(a) Original table.

T_a	f	v	c
	1	1	5
	6	2	7

T_b	f	v	c
	1	1	2
	3	2	3
	6	1	2
	8	3	5

T_c	f	v
	1	1
	2	4
	3	4
	4	5
	...	...

(b) Run-length encoded.

Figure 4.1: Run-length encoding in a row store. *Source:* [Bruno, 2009]

data layout comes at a much lower total cost of ownership (TCO), but is still able to serve a diverse range of workloads well enough. This is what FASE addresses.

The remainder of this chapter is organized as follows. After discussing related approaches in Section 4.1, we will illustrate the modeling of physical data layouts with the help of examples in Section 4.2. Based on these preliminaries, Section 4.3 defines the FASE notation, and Section 4.4 gives an overview of the architecture of FASE. We detail the implementation concepts of FASE in Section 4.5 and 4.6 before presenting evaluation results in Section 4.7. Section 4.8 summarizes the chapter.

4.1 Related Work

FASE aims at supporting a wider range of applications in a single system by offering a configurable specialization of the physical data layout. Most prominently, OLTP workloads favor a row-oriented data layout while OLAP workloads benefit from a column-oriented layout. Supporting these two workloads in a single system has been the aim of multiple projects in recent years. One group of approaches tries to improve the execution time of OLAP workloads on row stores by extending them with column-oriented query processing capabilities. Typically, newly developed hybrid row–column stores integrate both in a more balanced way. Some of these hybrid systems generalize the row store and column store layout into a configurable concept called column groups. A final group of related work aims more generally, like FASE, at increasing the physical data independence of database management systems. In the following, we discuss the different concepts and ideas in more detail.

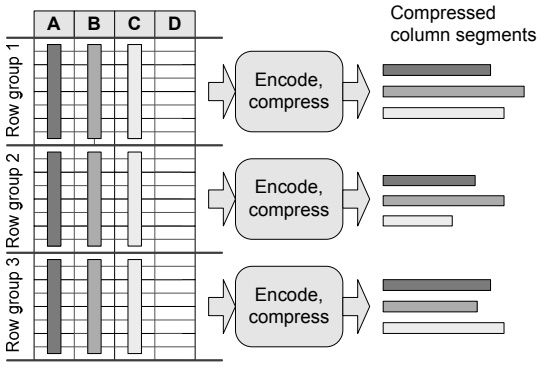


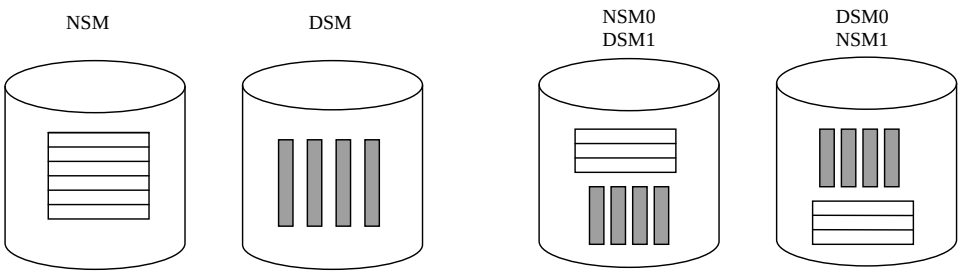
Figure 4.2: Column segment of column store index. *Source:* [Larson et al., 2011]

4.1.1 Column-Oriented Processing for Row Stores

The basic ways allow emulating column-oriented processing in a row store, as discussed in [Abadi et al., 2008]. The first way is to fully vertically partition a relation – originally described as the decomposed storage model [Copeland and Khoshafian, 1985]. Although very easy to implement, it is incomplete since every partition will include the primary key of the relation (or at least a surrogate key). For their superior performance, native column store implementations typically rely heavily on compression to increase cache utilization. Common compression techniques are run-length encoding [Schuegraf, 1976] and dictionary compression [Westmann et al., 2000]. Within the vertical partitioning approach run-length encoding can be emulated as well with the same performance benefits [Bruno, 2009]. The idea is illustrated in Figure 4.1. With this approach, we sort the original relation according to a given sorting schema, which encompasses all columns of the relation, and add a surrogate key (id). For each column, we generate a run-length encoded vertical partition. Each partition consists of the columns f, v, and c, where each tuple encodes a sequence of value v with the length c starting at the tuple with the surrogate f in the original relation. For instance, in the figure, the tuple $(1, 1, 5)$ in partition T_a encodes the sequence of five 1 in column a of the original relation running from tuple 1 to tuple 5. This approach considerably improves the performance of the vertical partitioning; however, it lacks the original simplicity.

The second way is index-only plans. In addition to the base relation, it adds an unclustered B+Tree index on every column. Containing rid–value pairs, the resulting indexes are similar to Binary Association Tables used by MonetDB [Boncz and Kersten, 1999]. With the help of index intersection, the database system can then answer queries without reading the base relation [Mohan et al., 1990, Raman et al., 2007]. Its efficiency can be further increased by exploiting the parallel processing capabilities of modern hardware [El-Helw et al., 2011].

Microsoft exploits the idea of column-oriented processing on indexes to the extreme by introducing a dedicated column store index [Larson et al., 2011]. A column store



(a) One physical layout per replica. (b) Multiple physical layouts per replica.

Figure 4.3: Fractured Mirrors. *Source:* [Ramamurthy et al., 2002]

index can be created on any number of columns of a relation. The index virtually partitions the relation into row groups resulting in one column segment per indexed column and row group, as shown in Figure 4.2. Each column segment is compressed and stored in a blob utilizing the existing blob storage mechanisms. A system table keeps the locations of the blobs and allows the database system to locate them quickly for query processing.

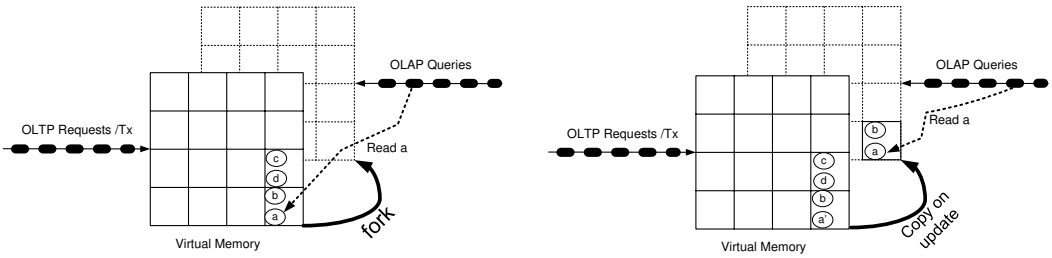
Trojan Columns [Jindal et al., 2013] follows the same line but omits the base relation completely. Here the data is also stored in blobs using the same kind of segmentation. However, the logical view of the relation is only emulated by user-defined table functions (UDFs). For query processing efficiency, the concept includes various UDFs which implement different levels of operator push-down.

From a practical standpoint, the column index seems the most appealing approach for implementing column-oriented processing in row stores. Here, the column processing is seamlessly integrated; it can barely be seen from the outside and reuses much of the existing functionality. Still, the column index remains an OLAP-focused add-on to a row-oriented database system.

4.1.2 Hybrid Row–Column Stores

Hybrid row–column stores try to integrate row and column orientation in a single engine to support OLTP and OLAP workload equally well. Hybrid engines can run solely OLTP or OLAP workloads, as well as mixed workloads efficently.

A very early approach of combining OLTP and OLAP is Fractured Mirrors [Ramamurthy et al., 2002]. Fractured Mirrors leverages the fact that disk-based databases usually replicate data to multiple disks, as illustrated in Figure 4.3. If the replicas hold the data in different physical layouts, queries can access the data in their preferred physical layouts. In the basic setup, the database system uses a single physical layout per replica (Figure 4.3(a)). Let us assume one replica uses the row store layout (NSM) and the other uses the column store layout (DSM). Aware of this setup, the query optimizer can easily route each query to its preferred physical data layout. The basic setup has two essential drawbacks. First, queries can only exploit

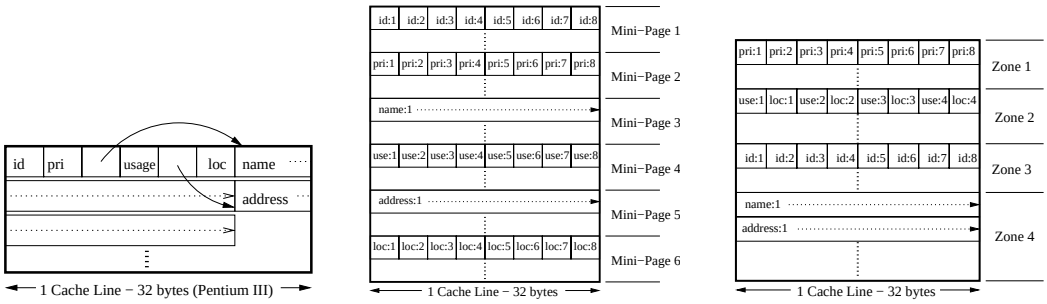


(a) New snapshot. (b) Copy on update.

Figure 4.4: Snapshot isolation in HyPer. *Source:* [Kemper and Neumann, 2011]

either the parallelism of the underlying hardware resources or exclusively use their preferred physical data layout, but not both at the same time. Second, workloads skewed towards one kind of physical data layout do not uniformly utilize the replicas. In the advanced setup, the database system uses multiple physical layouts per replica (Figure 4.3(b)). Given the system supports two phyiscal data layouts, the data is partitioned into two partitions and the partitions are replicated. The system makes sure that each replica of a partition has a different layout and that all partition replicas assigned to a hardware resource (a disk or a node) have a different layout. This allows exploiting the parallelism of the hardware resources, as well as a uniform utilization of these resources. As presented, Fractured Mirrors supports exactly the two representations, row store and column store. Both are implemented as different scan operators in the database engine. Generally, the idea is orthogonal to FASE. While the focus of FASE is to support various layouts within a single engine, Fracture Mirrors aims at exploiting replication to have the same data materialized in different layouts. Both concepts would make an appealing combination.

HyPer [Kemper and Neumann, 2011] is a main memory database system with OLTP and OLAP support. It runs OLAP queries concurrently and isolated from the OLTP queries, by utilizing hardware-supported page shadowing. More specifically, HyPer serially executes OLTP queries on the master copy of the data in a single process. To run OLAP queries in parallel and isolated from the OLTP workload, HyPer creates virtual memory snapshots that are transactional consistent with the OLTP queries. It does this by simply forking the OLTP process exactly between two OLTP queries. The virtual memory snapshot functionality of the operating system does not copy the complete address space of the OLTP process for the new OLAP process. Instead, the operating system reroutes memory access via virtual memory to the original physical memory block as shown in Figure 4.4(a). In case a concurrent OLTP transaction updates a memory block, the operating system creates a physical copy to keep the snapshot consistent – illustrated in Figure 4.4(b). In combination with the sequential execution of OLTP operations, this isolation mechanism eliminates the need for locking or any other kind of concurrency control, resulting in outstanding transaction throughput and query response times. However, the physical data layout



(a) Row store. (b) PAX. (c) Data Morphing.

Figure 4.5: PAX and Data Morphing. *Source:* [Hankins and Patel, 2003]

is not the authors' main concern. HyPer uses the same physical layout for all data; globally configured to either row-oriented or column-oriented. Other physical data layouts are not supported.

SAP HANA [Färber et al., 2011, 2012] is another hybrid main memory database system for OLTP and OLAP. It builds on a *multi-engine query processing environment*. Next to a text processing engine and a graph processing engine, HANA features a relational engine. The relational engine combines SAP's row store database engine P*Time and SAP's column store engine TREX. The column store engine heavily uses compression and is highly optimized for OLAP and data mining workloads [Legler et al., 2006, Lemke et al., 2010]. HANA's academic sidekick [Schaffner et al., 2008, Plattner, 2009] uses MaxDB as row store engine. The setup, though, is the same: two relational engines wired to the same query processor. During table creation the DBA decides which engine manages the table. The physical organization of a table is transparent for SQL queries, which can seamlessly combine row- and column-oriented tables. The SAP HANA approach differs considerable from FASE. Instead of supporting a hard-coded set of physical data layouts, FASE aims at a freely configurable physical data layout implemented in a single engine.

Generally, the possible physical data layouts of hybrid row–column stores remain limited to either row or column orientation per database table. FASE wants to offer broad range of physical data layout instead.

4.1.3 Column Groups

Column groups generalize the idea of combining row store and column store in a hybrid database system. Here a database system partitions a table vertically into configurable column groups. Depending on the partitioning, the resulting physical layout resembles either a row store or a column store, or something in between.

Data Morphing was the first approach to use the idea of bundling columns that are typically accessed [Hankins and Patel, 2003]. It is an extension of PAX [Ailamaki et al., 2001]. Like PAX, Data Morphing addresses page-oriented database systems and

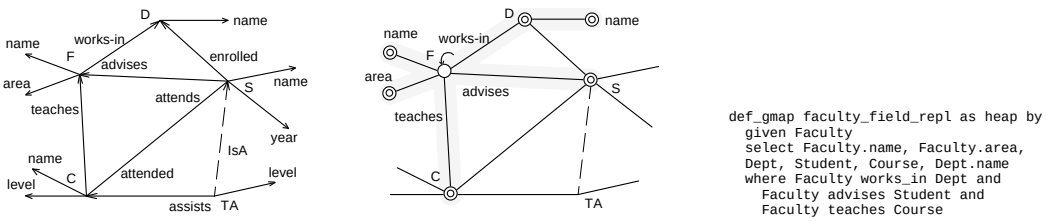
consideres the placement of values within a page. Figure 4.5 illustrates the difference of PAX and Data Morphing compared to the row store layout. The traditional row store layout subsequently places records in a page including all their values – depicted in Figure 4.5(a). Typically, fixed-length values come first and variable-length values are moved to the end. At the corresponding positions in the series of fixed-length values, a reference points to the actual position of the variable-length values. PAX organizes pages differently – depicted in Figure 4.5(b). For considerably better cache performance on OLAP queries, PAX lays out every page like a column store. For a table with six columns, a page is subdivided into six mini pages, each containing only the values of a single column. In the figure, the first mini page contains only id values, the second page only priority (pri) values and so on. While scanning data in a PAX page, queries that read only values of a single column receive considerably less cache misses because each cache line is packed only with values of interest. However, if multiple values of a record are accessed together, PAX's positive effect on cache performance disappears. Data Morphing improves that by exploiting column groups – depicted in Figure 4.5(c). Here, pages are organized in zones. Each zone stores one column group in the row store layout. If the column groups are designed properly, single column queries as well as multiple column queries will experience good cache performance.

HYRISE [Grund et al., 2010] is a main memory storage engine, which also makes use of column groups. However, it stores each column group dictionary compressed as if it is a single column. Although this is easy to implement, it could result in very large dictionaries, particularly if the columns in a column group are uncorrelated. Hence, at its heart, HYRISE is a main memory column store, which can merely be configured to mimic a row store or mixed layouts. As a distinctive feature HYRISE features an advisor that recommends a column group design for a given workload. Besides, HYRISE focuses on OLTP and OLAP workloads like HyPer and SAP HANA. Workloads favoring other physical data layouts are not considered.

4.1.4 Increased Physical Data Independence

While the majority of the related research work concentrates on the combination of OLTP and OLAP workloads, there is also research taking a broader hold on the topic and aiming at increased physical data independence. Physical data independence allows the DBA to adapt the physical representation of data to the performance requirements of the workload without influencing the logical model of the data [ANSI/X3/SPARC, 1975, Brodie and Schmidt, 1982]. This is a crucial feature of database management systems. However, traditional database systems do not support it to the full extent because their elementary physical data layout is typically hard-coded.

GENESIS [Batory, 1985, Batory et al., 1988] is an early project devoted to identifying common elements and structure in database storage systems. The project developed detailed storage models for database systems including aspects of the physical data layout. In GENESIS, a database storage system is visually modeled



(a) Logical schema. (b) Access path data. (c) Access path definition.

Figure 4.6: Gmap. *Source:* [Tsatalos et al., 1996]

with three kinds of diagrams: *data structure diagrams*, *field definition diagrams*, and *instance diagrams*. Files, links, records types, and fields are the building blocks of these diagrams. For instance, a data structure diagram defines files and how they are linked. Storage systems involve different levels of abstraction. Each level can be described with the diagrams, so that the diagrams form a hierarchical building plan of the described database storage systems. The aim of GENESIS was not to strengthen the physical data independence of database systems but to speed up their development. Consequently, GENESIS models physical data structures in a very detailed way. GENESIS' technical level of detail is that of database system developers rather than that of DBAs. FASE takes a different approach. Instead of providing an exhaustive description of a database system's storage layer, it focuses on capturing its macroscopic characteristics.

Another notable approach to increase physical data independence is Gmap [Tsatalos et al., 1994, 1996]. Gmap offers a physical data definition language for access paths. Where traditional access paths are fixed to the logical data model, Gmap defines the data stored in an access path with query-like statements. Figure 4.6 illustrates the concept by example. Given a logical database schema (Figure 4.6(a)), we can define an access path to store parts of the data (Figure 4.6(b)). The access path definition (Figure 4.6(c)) lists the name (`faculty_field_repl`), the type of data structure used (`heap`), the search key of the gmap structure (`Faculty`), and a query (`select ...`). The projection clause of the query determines which data field will be included in the access path (`Faculty.name, Faculty.area, Dept, ...`); the selection clause joins the data (`Faculty works_in Dept and Faculty ...`) and filters the data (not shown in the example). Gmap provides great flexibility in the physical design for the DBA. Next to the physical data definition language, the authors discuss query processing. They present algorithms to optimize queries on a gmap-structured physical storage so that the queries use a minimal set of benefical access paths. The focus of the gmap concept is to allow the consolidation of associated entities from different domains in a single access path. At the same time, gmap also allows the replication of entities over multiple access paths. Gmap offers a great way to define subsets of the data which require different physical data layouts. In that respect, Gmap and FASE complement each other perfectly.

RodentStore is the most recent work on physical data layout that we found [Cudré-Mauroux et al., 2009]. Like FASE, RodentStore envisions a freely configurable data layout, but with a different focus than FASE. RodentStore describes the physical data layout as nested lists of values. How the lists are nested and which values they contain can be configured with a storage algebra building on nested list comprehensions. Given a table *Product* with columns *name*, *price*, and *weight*, the list comprehension

$$[[r.name, r.price, r.weight] \mid \backslash r \leftarrow Product]$$

describes a row store data layout. In constrast, list comprehension

$$[[r.name \mid \backslash r \leftarrow Product], [r.price \mid \backslash r \leftarrow Product], [r.weight \mid \backslash r \leftarrow Product]]$$

describes a column store data layout. This allows a wide range of physical data layouts. With the focus rather on spatial data, RodentStore includes some representations not covered by FASE, such as the utilization of space filling curves. The RodentStore concept assumes strictly regular relational data, though, and does not consider the role of schema specifiers in the data representation. Hence, it lacks support for physical data layouts suitable for irregular data. Unfortunately, the presentation of RodentStore is rather visionary and lacks details on how queries and standard CRUD operations are processed in RodentStore. The prototype used by the authors for a case study appears to be rather limited. Nevertheless, it would be interesting to see how both concepts, RodentStore and FASE, can be combined.

4.2 Modeling Physical Data Layouts

Structured data consists of entities, which represent a set of attribute–value pairs each and are categorized into entity types. Physical storage, in contrast, is a consecutive one-dimensional sequence of bytes. The physical data layout represents the mapping between these two formats. Traditionally, DBMSs hardcode this mapping and the mapping is not configurable.

Commonly, structured data models organize values with the user-given specifiers entity types, entities, and attributes. A physical data layout defines how these elements of structured data (entity types, entities, attributes, and values) and the logical relations between them are stored physically. The most simple layout is a set of quadruples, illustrated in Figure 4.7. The layout is also know as vertical schema [Friedman et al., 1990, Agrawal et al., 2001, Cunningham et al., 2004]. A quadruple lists an entity type, an entity, and an attribute together with the value that belongs to the specific combination. In Figure 4.7, the first quadruple lists Smith as the value of the attribute *customer* for the entity o_1 with type *Order*. In the FASE notation, we denote the structure of such a quadruple as $T, E, A \rightarrow V$, where T is an element of the domain of entity types, E is an element of the domain of entities, A is an element of the domain of attributes, and V is an element of the domain of values. The arrow $\rightarrow$ indicates that T, E, A uniquely identify a quadruple and

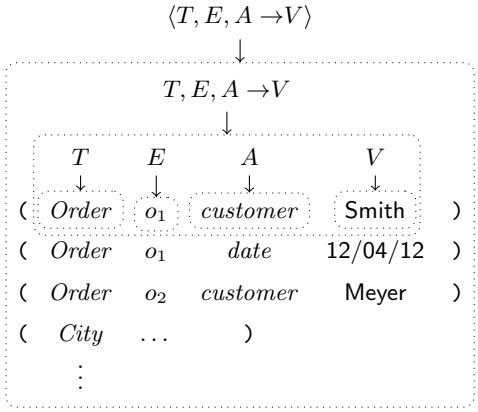


Figure 4.7: Data layout quadruples.

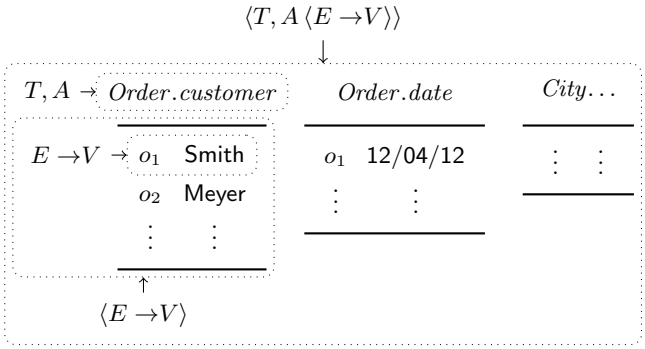


Figure 4.8: Data layout BATs.

functionally determine V in this mental model. To represent another value, we need another quadruple. For instance, in Figure 4.7, the second quadruple lists the value of the attribute *date* of *Order* o_1. Each value will have its own quadruple, resulting in a whole set of quadruples to represent a complete data set. In the FASE notation, we denote such a set of quadruples as $\langle T, E, A \rightarrow V \rangle$. Chevrons $\langle X \rangle$ indicate the repetition of the embedded structure X and denote that the order among instances of the embedded structure is insignificant. The commas in the FASE notation do not have a particular meaning but serve better visual separation of the domain symbols.

For obvious reasons quadruples are not commonly used as the physical data layout. As another example of physical data layouts let us have a look at binary association tables (BATs) instead. The columnar database system MonetDB uses BATs very successfully as physical data layout [Boncz and Kersten, 1999]. Figure 4.8 shows the same data represented in BATs. A database consists of multiple BATs and each BAT contains the values of a single attribute. In Figure 4.8, the first BAT from the left hosts all values belonging to the attribute *customer* of *Order* entities. Hence, we can denote the header of a BAT in the FASE notation as T, A. Each row of a

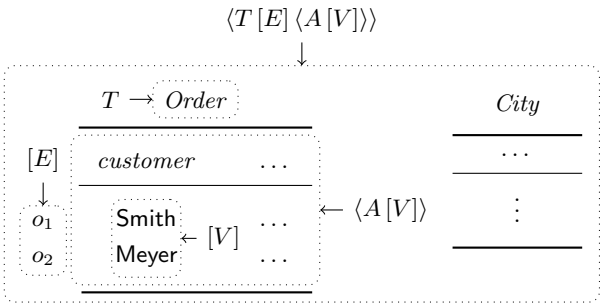


Figure 4.9: Data layout column store.

BAT contains an entity id and the corresponding value, where the entity id uniquely identifies the row. For instance, the first row in the first BAT of Figure 4.8 represents value Smith of attribute *customer* of *Order* o_1. In the FASE notation, the body of a BAT can be denoted as $\langle E \rightarrow V \rangle$. Correspondingly, a single BAT, consisting of header and body, is described with $T, A \langle E \rightarrow V \rangle$ in the FASE notation. The complete physical data layout used by MonetDB is then denoted as $\langle T, A \langle E \rightarrow V \rangle \rangle$.

While BATs explicitly associate entities and values with each other, many other columnar database systems rely on the physical order of the values in each column. We refer to this physical data layout as column store. Figure 4.9 shows the example data in the column store layout. The content of each column is physically organized as a list of values, where the physical order of the values matters. We denote this in the FASE notation as $[V]$. The brackets $[\ldots]$ indicate the repetition of the embedded structure and denote that the order among instances of the embedded structure is significant. A complete table of the column store layout consists of multiple columns including the attribute as column header. We denote this as $\langle A [V] \rangle$. The value lists in all columns of a table are ordered such that values belonging to the same entity appear at the same position. Additionally, the column store requires to know which position belongs to which entity, cf. Figure 4.9. We denote this in the FASE notation with a list of entity ids $[E]$, which is also part of the table. The entity type forms the table header, so that we can describe the column store data layout completely as $\langle T [E] \langle A [V] \rangle \rangle$.

The physical data layout of traditional row stores is pretty similar, as shown in Figure 4.10. Here, the rows of a table are represented with an entity id and an ordered list of values. As positional counterpart to the value lists, each table has an ordered list of attributes. The entity type forms the table header, so that the row store data layout denotes as $\langle T [A] \langle E [V] \rangle \rangle$.

4.3 FASE Notation

The FASE notation allows describing various physical data layouts for structured data, specifically the macroscopic aspects of the physical data layouts. The last section

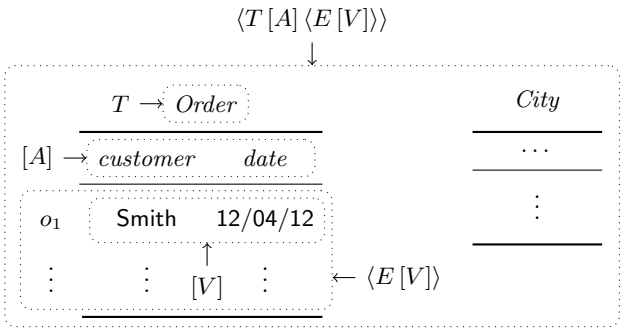


Figure 4.10: Data layout row store.

```
<collection> ::= <ordered> | <unordered>
   <ordered> ::= '[' <domain> ']'
 <unordered> ::= '⟨' <header> <include>? <nesting>? '⟩'
    <header> ::= <domain>+
   <include> ::= '→' <domain>+
   <nesting> ::= <collection>+
   <domain> ::= <specifier> | <values>
 <specifier> ::= 'T' | 'E' | 'A'
    <values> ::= 'V'
```

Figure 4.11: FASE grammar.

already illustrated the notation with the example of four data layouts. We will now discuss the grammar and the semantics of the FASE notation in detail.

A FASE expression describes a particular physical data layout, i.e. the schema in which a data set should be arranged physically on storage. To be valid, a FASE expression has to follow the grammar shown in Figure 4.11. The central element is the collection definition (<collection>). A collection can be defined as ordered (<ordered>) or as a collection without particular order (<unordered>). Ordered collections are defined on single domains. Further, ordered collections are only allowed in pairs in FASE expressions, where one of the two ordered collections has to be defined on the domain of values V. Normal collection definitions consist of header domains (<header>), included domains (<include>), and nested collections (<nesting>). There must be at least one header domain; included domains and nested collections are optional. Among the four domains, we distinguish between the specifier domains – entity types T, entities E, and attributes A – and the values domain V. We are not considering values to have an identity of their own; value identity comes from the entity type, the entity, and the attribute a value belongs to. Consequently, V is only allowed in ordered collections or as an included domain in unordered collections.

When parsed, a FASE expression results in an abstract syntax tree of collection definitions as shown in the upper half of Figure 4.12. These collection definitions

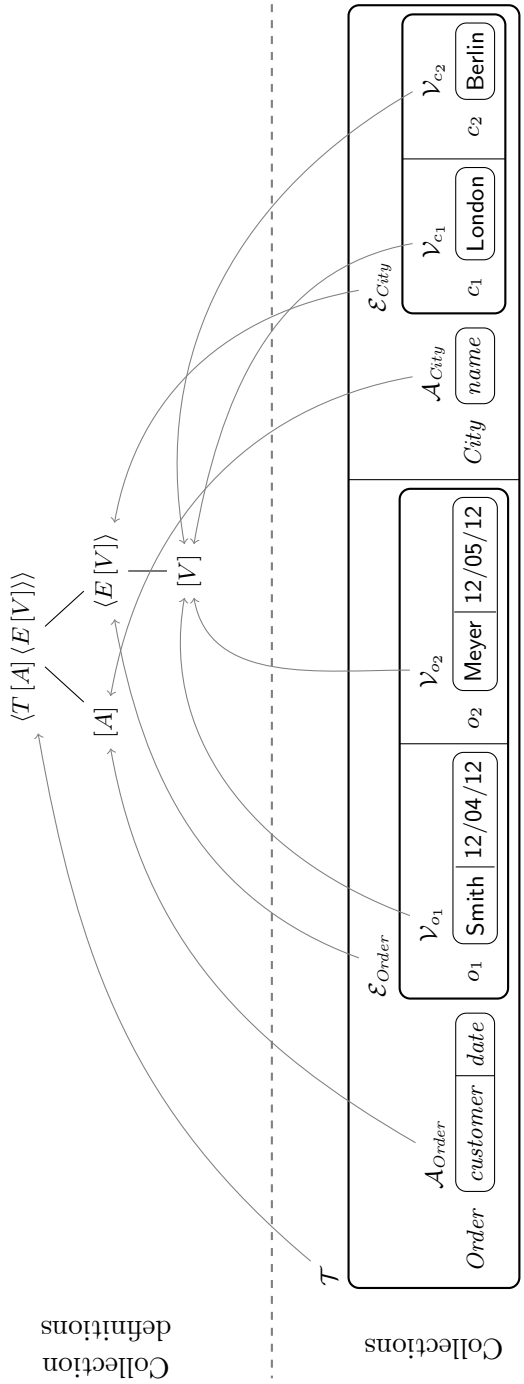


Figure 4.12: Collection with the row store layout $\langle T\,[A]\,\langle E\,[V]\rangle\rangle$.

provide the necessary information to physically arrange and retrieve data from storage. Like the collection definitions, the physical layout described by the expression is hierarchical. At each hierarchy level the collection definition is instantiated with one or more collections, so that each collection is typed by a particular collection definition from the abstract syntax tree. The topmost collection definition is only instantiated once, while lower-level collection definitions are likely to be instantiated multiple times, depending on the data in the system.

Collections, the instances of collection definitions, contain one or multiple groups. The groups contain data elements and may nest lower-level collections. The collection definition of a collection defines the internal structure of its groups. For instance, in a collection of the definition $\langle E\,[V]\rangle$, all groups take the form $(e, \mathcal{X})$, where e is an entity and $\mathcal{X}$ is a collection of the definition $[V]$. All groups in a collection are uniquely identified by their headers, i.e. their elements from the header domains in the definition. In the example, all groups are uniquely identified by the data element e. A group's data elements from the included domains in the collection's definition form the group include.

As an example, Figure 4.12 illustrates the collections resulting from the row store layout $\langle T\,[A]\,\langle E\,[V]\rangle\rangle$ given our example data. In the figure, the rounded rectangles mark collections, while the vertical lines separate groups within collections. The topmost collection $\mathcal{T}$ is an instance of $\langle T\,[A]\,\langle E\,[V]\rangle\rangle$ as indicated by the arrow pointing from the collection to its collection definition. Consequently, it contains groups consisting of an entity type as header and two nested collections of the form $[A]$ and $\langle E\,[V]\rangle$. $\mathcal{T}$ contains two such groups: $(Order, \mathcal{A}_{Order}, \mathcal{E}_{Order})$ and $(City, \mathcal{A}_{City}, \mathcal{E}_{City})$. The order of these two groups as given in the figure is insignificant. Further down in the hierarchy, for instance, the collection $\mathcal{E}_{Order}$ is of the form $\langle E\,[V]\rangle$ and contains the groups $(o_1, \mathcal{V}_{o_1})$ and $(o_2, \mathcal{V}_{o_2})$. $\mathcal{V}_{o_1}$ and $\mathcal{V}_{o_2}$ are both ordered collections of the form $[V]$, each containing a pair of groups, (Smith), (12/04/12) and (Meyer), (12/05/12), respectively. Here, the order of the groups is essential since it allows connecting the values to their attributes stored in the ordered collection $\mathcal{A}_{Order}$.

Table 4.1 lists a sample of physical data layouts that can be denoted with the FASE notation. Next to the layouts already discussed in Section 4.2, the table lists the interpreted record layout [Beckmann et al., 2006, Chu et al., 2007], the Bigtable layout [Chang et al., 2006, 2008] and a layout that organizes the data like XML tags [W3C, 2008]. Each data layout is shown with its respective FASE expression and an example of the resulting collections.

4.4 FASE Architecture

FASE, as a storage engine, implements a configurable physical data layout that can be specified using the FASE notation. Figure 4.13 shows its principal architecture. At the bottom, FASE builds on a physical record interface that provides record containers. At the top, FASE provides a logical data interface with basic CRUD operations. FASE itself is two-layered. It consists of a collection formation layer and a collection

Table 4.1: Various physical data layouts.

Data layout	Layout definition	Example
Quadruple	$\langle T, E, A \rightarrow V \rangle$	$\langle Order, o_1, customer \rightarrow \mathsf{Smith} \mid Order, o_1, date \rightarrow \mathsf{12/04/12} \mid Order, o_2, customer \rightarrow \mathsf{Meyer} \mid \ldots \rangle$
Row store	$\langle T\,[A]\,\langle E\,[V] \rangle \rangle$	$\langle Order, [customer \mid date]\,\langle o_1, [\mathsf{Smith} \mid \mathsf{12/04/12}] \mid o_2, [\mathsf{Meyer} \mid \mathsf{12/05/12}] \rangle \rangle$
Column store	$\langle T\,[E]\,\langle A\,[V] \rangle \rangle$	$\langle Order, [o_1 \mid o_2]\,\langle customer, [\mathsf{Smith} \mid \mathsf{Meyer}] \mid date, [\mathsf{12/04/12} \mid \mathsf{12/05/12}] \rangle \rangle$
BATs	$\langle T, A\,\langle E \rightarrow V \rangle \rangle$	$\langle Order, customer\,\langle o_1 \rightarrow \mathsf{Smith} \mid o_2 \rightarrow \mathsf{Meyer} \rangle \mid Order, date\,\langle o_1 \rightarrow \mathsf{12/04/12} \mid o_2 \rightarrow \mathsf{12/05/12} \rangle \rangle$
Interpreted record	$\langle T\,\langle E\,\langle A \rightarrow V \rangle \rangle \rangle$	$\langle Order\,\langle o_1, \langle customer \rightarrow \mathsf{Smith} \mid date \rightarrow \mathsf{12/04/12} \rangle \mid o_2, \langle customer \rightarrow \mathsf{Meyer} \mid date \rightarrow \mathsf{12/05/12} \rangle \rangle \rangle$
Bigtable	$\langle T\,\langle E, A \rightarrow V \rangle \rangle$	$\langle Order\,\langle o_1, customer \rightarrow \mathsf{Smith} \mid o_1, date, \rightarrow \mathsf{12/04/12} \mid o_2, customer \rightarrow \mathsf{Meyer} \mid \ldots \rangle \rangle$
XML-like	$\langle E \rightarrow T\,\langle A \rightarrow V \rangle \rangle$	$\langle o_1 \rightarrow Order\,\langle customer \rightarrow \mathsf{Smith} \mid date \rightarrow \mathsf{12/04/12} \rangle \mid o_2 \rightarrow Order\,\langle customer \rightarrow \mathsf{Meyer} \mid \ldots \rangle \rangle$

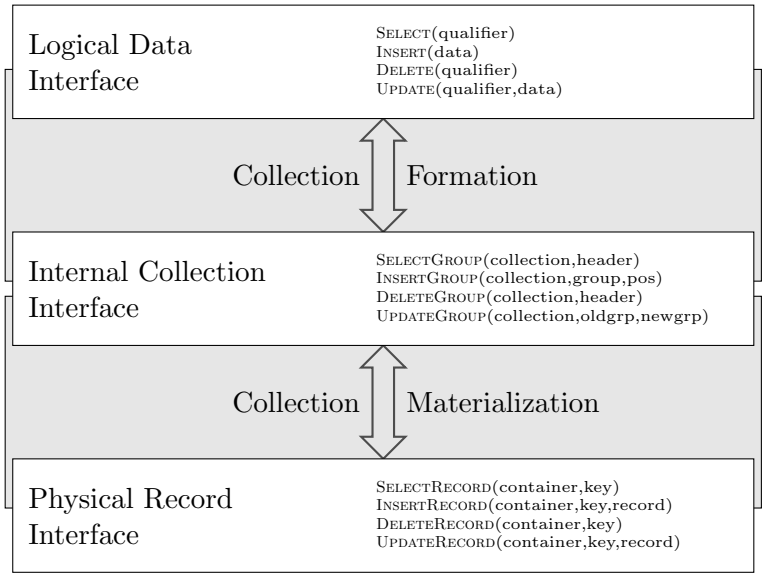


Figure 4.13: FASE architecture.

materialization layer. Collection formation organizes the data in the hierarchical layout specified by a FASE expression. Collection materialization maps the collections and the groups they contain to containers and records. Internally, the two layers are separated by the collection interface. We will first discuss the interfaces from the bottom up, and then elaborate on collection formation and collection materialization in the following two sections in more detail.

Physical Record Interface The physical record interface provides the basic storage routines for FASE. It offers storage containers for records, i.e. byte strings. Containers can be created and deleted via the interface. The key is either provided together with the record or generated by the container. The record interface allows creating and dropping containers as well as selecting (SELECTRECORD), inserting (INSERTRECORD), deleting (DELETERECORD), and updating (UPDATERECORD) records in a given container.

Internal Collection Interface Collections of groups provide the central data representation format during processing within FASE. The collection interface allows operating with collections and groups. Collections can be created and dropped. On a given collection, four major operations allow handling the groups of the collection. (1) SELECTGROUP retrieves all groups from a collection that have a header matching a given header predicate. Header predicates have the same elements as the header, but additionally allow wildcards, such as '∗', which matches any element. (2) INSERTGROUP appends a new group to a collection. For ordered collections, a numeric index can be specified to insert the group at a specific position.

(3) DeleteGroup removes all groups with a header matching a given header predicate. Ordered collections also allow positional deletions. (4) UpdateGroup updates a given group.

Logical Data Interface The logical data interface offers fundamental CRUD operations to manage data values with their specifiers: entity types, entities, and attributes. The user can *Select*, *Insert*, *Delete*, and *Update* data in a database.

Select retrieves values from the database. Provided with a qualifier triple (t, e, a), a select retrieves only the values that have specifiers matching the given triple. The wildcard '$*$' matches any element. For instance, $(Order, *, *)$ retrieves all values of all entities of the entity type *Order*. $(Order, o_i, *)$ yields the values of the particular order o_i, whereas $(Order, *, date)$ returns all values belonging to attribute *date* for all entities of entity type *Order*.

Insert adds data to a database. Given a quadruple (t, e, a, v), an insert adds each value v for the given specifiers t, e, and a. An insert also accepts a list of such quadruples. In case a list of quadruples is given, successively reoccurring specifiers can be replaced with '%' for simplicity. For example, the list $(Order, o_1, customer, \mathsf{Smith})$, $(\%, \%, date, \mathsf{12/04/12})$, $(\%, o_2, \text{customer}, \mathsf{Meyer})$, $(\%, \%, date, \mathsf{12/05/12})$ inserts the data shown in Table 4.1.

Delete removes values from the database. Similar to select, a delete takes a qualifier triple and deletes all values with matching specifiers. For instance, $(Order, *, *)$ deletes all values belonging to entity type *Order*. $(Order, o_i, *)$ deletes the particular order o_i, and $(Order, *, date)$ would delete all values belonging to attribute *date*.

Update changes values in the database. Given a qualifier triple, an update changes all values with matching specifiers to a given data quadruple. The wildcard '$*$' is allowed in the qualifier triple. In the target quadruple, '%' indicates that matching data elements should remain unchanged. Note that a change can affect a value itself or its specifiers. For instance, an update $(Order, o_1, date) \rightarrow (\%, \%, \%, \mathsf{12/05/12})$ changes the date of order o_1 to the 5th December. Whereas, $(Order, *, date) \rightarrow (\%, \%, orderdate, \%)$ renames attribute *date* for entity type *Order* to *orderdate*.

To achieve data independence in FASE, the logical data interface functionally behaves irrespectively of the chosen data layout. However, the interface's non-functional behavior, i.e. the performance of operations, depends strongly on the physical data layout in use. For instance, a column-oriented access, such as $(Order, *, date)$ will be favored by a column-oriented physical layout such as $\langle T\,[E]\,\langle A\,[V]\rangle\rangle$. This way, FASE allows optimizing of the database performance, without changing existing application code or even the query processing infrastructure of a database management system.

To summarize FASE's architecture: Collection formation translates the operations of the data interface to collection operations. Collection materialization translates collection operations to record container operations. In the following two sections, we

will present in detail how these translations are done in the corresponding parts of FASE.

4.5 Collection Formation

Collection formation forms the upper layer in the archtiecture of FASE. It translates the operations of the logical data interface into operations of the collection interface. The general procedure for all operations starts with the topmost collection and recursively treats collections nested in groups. The following subsections detail the algorithm for each of the four operations of the logical data interface.

4.5.1 Select

The select algorithm performs two tasks. First, it retrieves the queried values and their specifiers. Second, it reassembles the data back into the representation-agnostic format of the logical data interface. Both tasks are done in a single run.

Retrieval is simple. The algorithm filters each collection for the groups that match the query predicate. It is important that the filtering is done before the algorithm reads nested collections to avoid unnecessary reading collections. The filter predicate of a collection is obtained by projecting the query predicate to the collection's header.

Reassembling the data is more complex and done in three steps. First, if a group has multiple nested collections, the matching groups found in the nested collections have to be joined. If ordered collections are involved, groups are joined by shared domains and their order. Second, group header and group include are added to each of the groups resulting from the join. Conceptually, this is a cross product. Finally, the results from all groups in a collection are united into a single result set.

Figure 4.14 illustrates the reassembling process by example. The figure shows the query $(Order, *, *)$, which retrieves all entities of entity type *Order* with all their attributes. It is executed on the example data used throughout this chapter. The data is organized according to the FASE expression $\langle T\,[A]\,\langle E\,[V]\rangle\rangle$, which reassembles a row store. Considering only the *Order* part of the data, the data layout results in the five collections $\mathcal{T}$, $\mathcal{R}_{Order}$, $\mathcal{C}_{Order}$, $\mathcal{V}_{o_2}$, and $\mathcal{V}_{o_2}$ similar to Figure 4.12. The reassembling process runs bottom up. As can be seen, the algorithm combines the groups of the most nested collections $\mathcal{V}_{o_1}$ and $\mathcal{V}_{o_2}$ with the header of their respective parent group from collection $\mathcal{E}_{Order}$. For instance, the two groups $(1, \mathsf{Smith})$ and $(2, \mathsf{12/04/12})$ from the collection $\mathcal{V}_{o_1}$ are combined with (o_1) to $(1, o_1, \mathsf{Smith})$ and $(2, o_1, \mathsf{12/04/12})$. In the process, the algorithm maintains the order of the groups from the ordered collections $\mathcal{V}_{o_1}$ and $\mathcal{V}_{o_2}$, illustrated by the integers preceding the data elements in the groups. Next, the results of the two combinations are united to form the result for the collection $\mathcal{E}_{Order}$. The collection $\mathcal{E}_{Order}$ and its sibling collection $\mathcal{A}_{Order}$ are nested in the same group. Their respective results are ordered and do not share any domains. Hence, the algorithm joins the results of $\mathcal{E}_{Order}$ and $\mathcal{A}_{Order}$ according to their order. The group $(1, customer)$ pairs with the groups $(1, o_1, \mathsf{Smith})$ and $(1, o_2, \mathsf{Meyer})$ and so on. Afterwards, the algorithm combines the result of the join

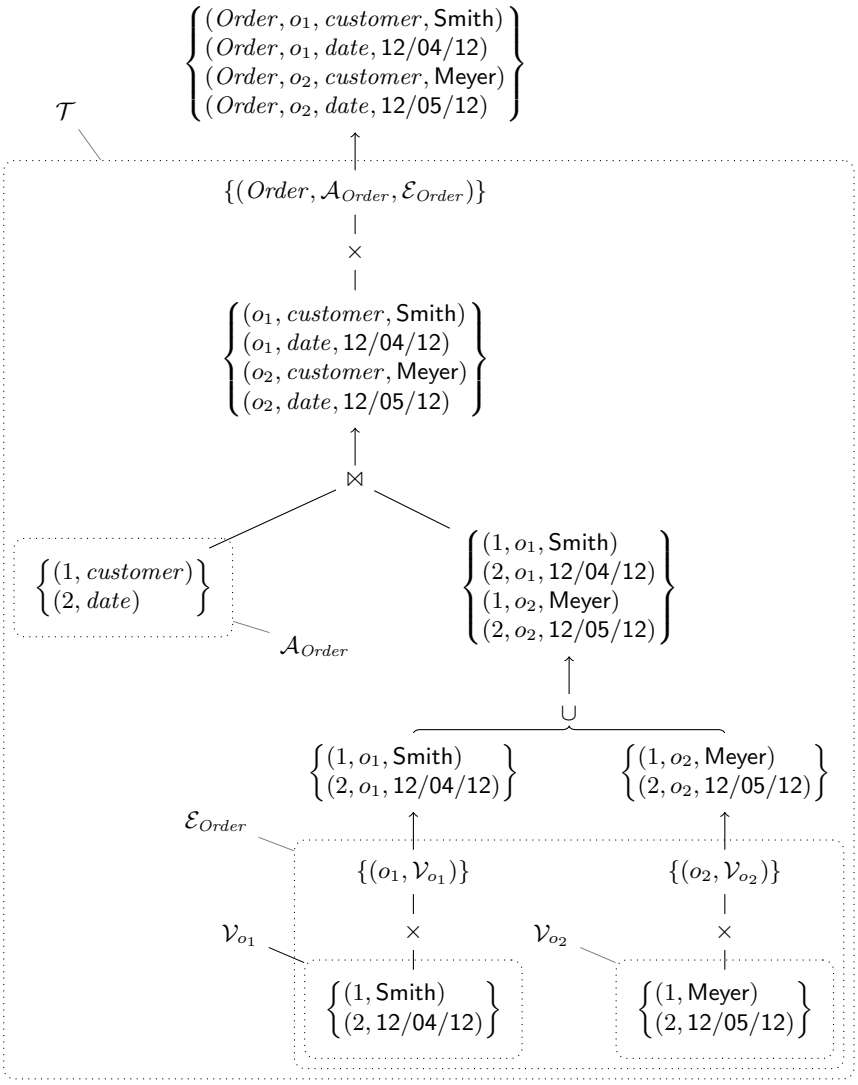


Figure 4.14: Query $(Order, *, *)$ on $\langle T\,[A]\,\langle E\,[V]\rangle\rangle$.

with the header of the nesting group $(Order, \ldots)$. Since this group is in the topmost collection $\mathcal{T}$, the reassembling is complete.

Algorithm 2 summarizes the recursive select procedure performed on every collection $\mathcal{S}$. First, the algorithm filters the collection (lines 4–7). Therefore, it determines the header predicate h by projecting the qualifier triple q to the collection's header domains. Here, $\mathcal{S}$.H and $\mathcal{S}$.I represent the header domains and included domains of the collection $\mathcal{S}$, respectively. Second, the algorithm retrieves all matching groups from the collection. Additionally, all qualifying groups are filtered for matching includes. Naturally, this filtering is only effective if the groups in the current collection have includes. Third, the algorithm performs the reassembling routine for every qualifying

Algorithm 2 Select from collections.

```
1:  procedure SELECT(S, q)                                  ▷ S: collection to query
2:                                                          ▷ q: qualifier triple (t, r, c)
3:      R_S ← ∅                                             ▷ init collection result
4:      h ← π_{S.H}(q)                                      ▷ project qualifier to header
5:      l ← π_{S.I}(q)                                      ▷ project qualifier to include
6:      for all G ∈ SELECTGROUP(S, h) do                    ▷ find group
7:          if G.I matches l then                           ▷ filter on include
8:              if G.N ≠ ∅ then                             ▷ if group has nestings
9:                  R_J ← ⋈_{U∈G.N} SELECT(U, q)            ▷ join
10:                 R_G ← ({G.H} × {G.I}) × R_J             ▷ combine
11:             R_S ← R_S ∪ R_G                             ▷ unite
12:     return R_S                                          ▷ return collection result
```

group (lines 8–11): (1) join results from nested collections (line 9), (2) combine with group header and group include (line 10), and (3) unite with results from other groups of the same collection (line 11). Obviously, if a group does not have any nested collections, the first two steps are omitted (line 8). Finally, the algorithm returns the collection result.

Although join, cross product, and union seem to be expensive operations, FASE implements the reassembling process efficiently. The join of ordered results is implemented as a sort-merge join. Joining shared domains requires a hash join. The combination of the join result with group header and group include is efficient too, since the operation always has a single group as its left operand. Further, the uniqueness of group headers within a collection guarantees that the group results are free of duplicates. Hence, the union simply appends the group results to the collection result and expensive duplicate elimination is not required.

4.5.2 Insert

The insert algorithm arranges the data as described by the used FASE expression. Depending on what should be inserted and which data is already in the database, the algorithm has to create new groups or even new collections. We discuss unordered and ordered collections separately.

Let us consider normal (unordered) collections first. Unordered collections either already contain a group that matches the data to insert or they do not. In the first case, the insert algorithm selects the matching group and propagates the insert operation to all collections nested in the discovered group. In the second case, the insert algorithm creates a new group and adds it to the collection. If the new group has to have nested collections, these are created as well as and the insert is propagated to them.

As an example, let's consider the normal collection $\mathcal{T}$ as shown in Figure 4.12. For the insert $(\mathit{City}, c_3, \mathit{name}, \mathit{New\ York})$, the algorithm selects the group

$(City, \mathcal{A}_{City}, \mathcal{E}_{City})$ from $\mathcal{T}$ and recursively propagates the insert to the two nested collections, $\mathcal{A}_{City}$ and $\mathcal{E}_{City}$. In contrast, for inserting $(Supplier, s_3, name, Tools\ Inc.)$, the algorithm appends a new group $(Supplier, \mathcal{A}_{Supplier}, \mathcal{E}_{Supplier})$ to $\mathcal{T}$. It also creates the two new collections, $\mathcal{A}_{Supplier}$ and $\mathcal{E}_{Supplier}$ and propagates to them.

Algorithm 3 Insert into unordered collections.

```
 1: procedure INSERT(S, d)                                  ▷ S: collection
 2:                                                         ▷ d: data quadruple (t, r, c, v)
 3:     h ← π_{S.H}(d)                                      ▷ project data to header
 4:     l ← π_{S.I}(d)                                      ▷ project data to include
 5:     G ← SELECTGROUP(S, h)                               ▷ find matching group
 6:     if G exists then
 7:         check l = G.I                                   ▷ check for matching include
 8:     else
 9:         N ← {U | U = CREATECOL(T) ∧ T ∈ S}
10:         G ← (h, l, N)                                   ▷ create new group
11:         INSERTGROUP(S, G)                               ▷ insert into collection
12:     for all U ∈ G do                                    ▷ propagate insert
13:         INSERT(U, d)                                    ▷ to nested collections
```

Algorithm 3 lists the basic insert algorithm for unordered collections. First, the algorithm projects the data that should be inserted to the header and the included domains of the current collection (lines 3–4). Then, the insert algorithm tries to find a matching group (line 5). If a matching group G exists in the collection, the algorithm checks if the include of G matches the data to insert (line 7). To avoid storing the value with wrong specifiers, the insert is declined with an error if the include does not match. If a matching group does not exist in the collection, the algorithm creates a new group G and adds it to the collection (lines 9–11). This involves creating nested collections in the group if the used FASE expression requires so. By now, the data to insert in the current collection is definitely present in group G; it either already existed or has been created. Finally, the insert algorithm propagates the insert operation to collections nested in G (lines 12–13).

Ordered collections are more complicated to handle. Valid FASE expressions require pairs of ordered collections, where one of the two ordered collections is defined on the values domain V and the other on a specifier domain. We refer to them as value collection and specifier collection, respectively. While adding new data, the insert algorithm has to keep the order of a pair of ordered collections in sync. It achieves this by handling them in pairs, as if they were a single collection. Thus, the specifier collection defines the shared order and the value collection follows it.

Let's assume that the two ordered collections are $\mathcal{A}_{Order} = [customer \mid date]$ and $\mathcal{V}_{o_1} = [\mathsf{Smith} \mid 12/04/12]$, as shown in Figure 4.12, and the insert request is $(Order, o_1, status, \mathsf{finished})$. To keep both ordered collections in sync, the insert algorithm adds *status* to the specifier collection $\mathcal{A}_{Order}$ at the first free position (in the

example, third position in the list). Then the algorithm inserts the value finished in $\mathcal{V}_{o_1}$ at exactly the same position. The insert results in $\mathcal{A}_{Order} = [customer \mid date \mid status]$ and $\mathcal{V}_{o_1} = [\text{Smith} \mid 12/04/12 \mid \text{finished}]$.

Note that the algorithm has to keep $\mathcal{A}_{Order}$ and $\mathcal{V}_{o_2} = [\text{Meyer} \mid 12/05/12]$ in sync, too. After the first insert, $\mathcal{A}_{Order}$ is one element longer than $\mathcal{V}_{o_2}$. Still, both collections are in sync because all values in $\mathcal{V}_{o_2}$ are correctly related to their specifiers. Let's assume a second insert operation $(Order, o_2, status, \text{open})$. Again, the insert algorithm handles $\mathcal{A}_{Order}$ first. Since *status* is already in $\mathcal{A}_{Order}$ the algorithm only takes its position to insert the value open in $\mathcal{V}_{o_2}$. Accordingly, the result of that second insert is $\mathcal{V}_{o_2} = [\text{Smith} \mid 12/05/12 \mid \text{open}]$.

An insert may lead to gaps in a value collection. For instance, assume the second insert operation is $(Order, o_2, priority, \text{high})$. This would result in $\mathcal{A}_{Order} = [customer \mid date \mid status \mid priority]$ and $\mathcal{V}_{o_2} = [\text{Smith} \mid 12/05/12 \mid \square \mid \text{high}]$, where $\square$ marks the new gap. The group materialization has to handle these gaps appropriately so that the positions of the values remain unaffected.

Ordered collections are handled in pairs. While recursively descending the hierarchy of nested collections, the insert algorithm postpones the handling of an ordered collection until it passes its mate. In the example of Figure 4.12, the algorithm postpones the handling of $\mathcal{A}_{Order}$ until it reaches $\mathcal{V}_{o_1}$ or $\mathcal{V}_{o_2}$. During postponing, the algorithm simply keeps a reference to the respective collection. Once a pair of ordered collections is found, the data is inserted appropriately. After the actual insert, the algorithm keeps the reference to the specifier collection to pair it with the other value collections. In the example, after processing the pair $\mathcal{A}_{Order}$–$\mathcal{V}_{o_1}$, the algorithm keeps $\mathcal{A}_{Order}$ for handling $\mathcal{V}_{o_2}$.

Algorithm 4 Insert into ordered collections.

```
 1: procedure INSERT(S_S, S_V, d)                    ▷ S_S: specifier collection
 2:                                                  ▷ S_V: value collection
 3:                                                  ▷ d: data quadruple (t, e, a, v)
 4:     h_S ← π_{S_S.H}(d)                          ▷ project data to S_S
 5:     h_V ← π_{S_V.H}(d)                          ▷ project data to S_V
 6:     G ← SELECTGROUP(S_S, h_S)                   ▷ find matching group
 7:     if G not exists then                        ▷ if group not found
 8:         G ← (h_S)                               ▷ create new specifier group
 9:         INSERTGROUP(S_S, G)                     ▷ insert into collection
10:     check that G.Pos is free in S_V             ▷ check position
11:     INSERTGROUP(S_V, (h_V), G.Pos)              ▷ insert value
```

Algorithm 4 shows the insert into a pair of ordered collections. After projecting the data to the respective collections (lines 4–5), the algorithm looks for a matching group in the specifier collection (line 6). If none is found, an appropriate group is created and inserted to the specifier collection (lines 7–9). Afterwards the algorithm checks if the position that the specifier group has in the corresponding specifier collection is

free in the value collection and inserts a new value group at that position (lines 10–11). If the position is not free, the insert is rejected since it represents a duplicate.

4.5.3 Delete

The delete algorithm removes all values that have specifiers matching the given qualifier. Therefore, the algorithm descends the hierarchy of nested collections to the values in question by finding the matching group on each level. Once the values are found, the algorithm deletes the groups containing them.

Furthermore, the algorithm also has to remove orphaned specifiers, which are not related to a value anymore. This is done when the algorithm climbs back up the hierarchy of nested collections. If a group is deleted from a collection, the algorithm checks if the collection is empty. If so, it deletes the empty collection as well. As a consequence, the algorithm can also delete the group that contained the empty collection. This procedure will remove all orphaned specifiers from unordered collections.

Ordered collections are trickier regarding orphaned specifier removal. With ordered collections, the groups containing the specifiers do not nest the groups containing the related values. It does not become implicitly clear whether a specifier is orphaned or is still needed. For instance, to delete the specifier *date* in $\mathcal{A}_{Order}$, the algorithm has to ensure that both collections, $\mathcal{V}_{o_1}$ and $\mathcal{V}_{o_2}$ have no values at the corresponding position. Scanning all value collections related to a single specifier collection can become very costly. In comparison, the chances that a specifier actually is orphaned are rather low. Hence, FASE does not delete orphaned specifier in ordered specifiers collections implicitly. They have to be deleted explicitly by the user.

4.5.4 Update

An update operation can affect values, their specifiers, or both. The general procedure for updates is to delete all qualifying data and reinsert the updated version. Naturally, the algorithm retrieves the qualifying data during the delete to obtain the updated version. However, this delete–insert pattern can result in unnecessary data movement, e.g. if only an entity type is renamed. FASE's update algorithm tries to avoid the basic delete–insert pattern where possible.

The extent to which data movement can be avoided depends on the FASE configuration. For instance, consider the update $(Order, *, date) \rightarrow (Timeline, \%, \%, \%)$, which moves all values belonging to the attribute *date* to another entity type. In the rows store layout $\langle T\,[A]\,\langle E\,[V]\rangle\rangle$, this update requires deleting all corresponding values from the $\mathcal{R}_{Order}$ collections within the $(Order, \mathcal{A}_{Order}, \mathcal{E}_{Order})$ group and inserting these values in the $\mathcal{R}_{Timeline}$ collection of a $(Timeline, \mathcal{A}_{Timeline}, \mathcal{E}_{Timeline})$ group. The same has to be done for the specifier *date*. In contrast, the BAT layout $\langle T, A\,\langle E \rightarrow V\rangle\rangle$ requires only an in place update of *Order* to *Timeline* in the $(Order, date, \ldots)$ group of the topmost collection.

To avoid unnecessary data movement, the update algorithm diverges from the general delete–insert pattern in situations formulated in the following three update optimization rules. (1) If the current collection has no nested collection that needs a direct update, the update algorithm stops its recursion. (2) If the current collection does not require a direct update but nested collections do, the update algorithm descends to all nested collections without any changes to the current collection. (3) If the current collection has no nested collection that requires a selection, the update algorithm changes the current collection in place.

Which collections require direct updates and selections can be determined from the qualifier triple and the target quadruple. A collection requires a direct update if one of its header domains and included domains is unequal to '%' in the target quadruple. Whereas, a collection requires a selection if one of its header domains and included domains is unequal to '$*$' in the qualifier triple. Note that the values domain V is not present in the qualifier triple, so it is implicitly equal to '$*$'.

Ordered collections require special attention during the updates. If a group has a nested ordered collection, the algorithm does not apply the first update optimization rule to all collections nested in this group. This is necessary to propagate the update to both sides of a pair of ordered collections. In-place updates on the specifier collection of a pair of ordered collections are only possible if the qualifier selects all value collections that pair with the specifier collection in question. Value collections are always updated in-place in case the update algorithm descended that far without previously having switched to the delete–insert pattern before.

4.6 Collection Materialization

Collection materialization forms the lower layer in the architecture of FASE, cf. Figure 4.13. It translates operations of the internal collections interface to operations of the physical record interface. The record interface provides logically unrelated containers for the storage of untyped records. In contrast, collections form a well-typed, hierarchically nested structure. There are two general ways to materialize nested collections in containers. In this section, we discuss both ways and present an automatic procedure used in FASE to switch between them.

4.6.1 Dedicated vs. Embedded

Dedicated materialization stores a collection in a dedicated container. Every group in the collection is mapped to a record. The topmost collection of a database is always stored using dedicated materialization. In contrast, embedded materialization stores a whole collection with all its groups embedded in the record that materializes the group that nests the collection. Naturally, embedded materialization is only possible for nested collections.

Figure 4.15 shows the collection materialization of the example data using the FASE configuration $\langle T\,[A]\,\langle E\,[V]\rangle\rangle$. In the figure, the upper half shows the data transformed into collections and groups similar to Figure 4.12, while the lower half

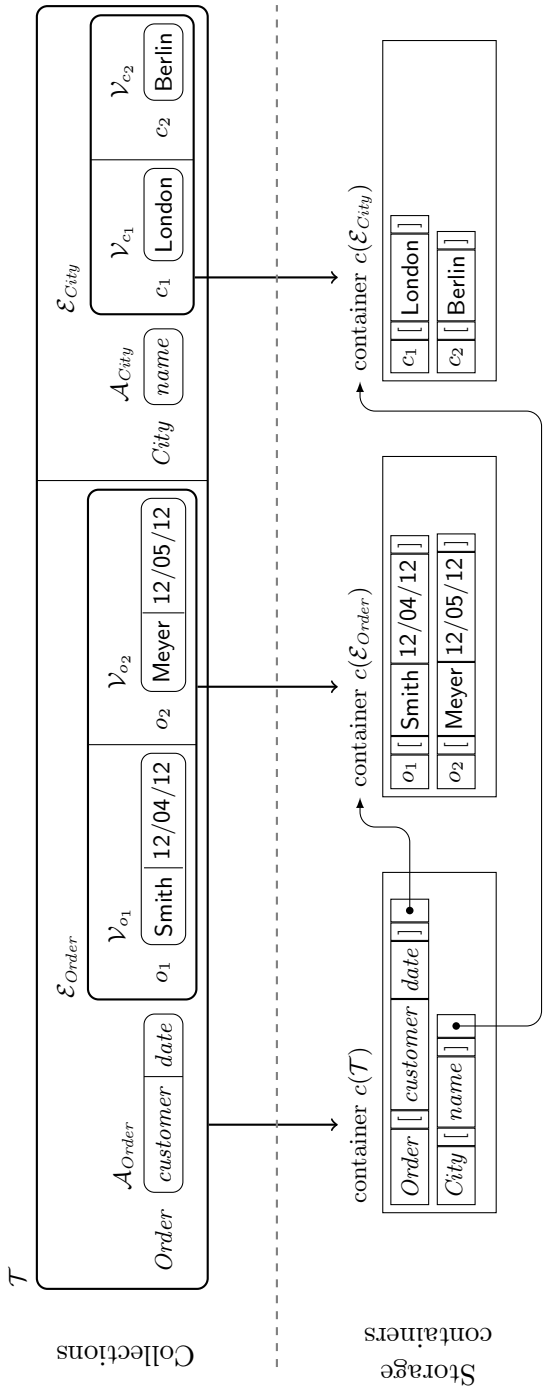


Figure 4.15: Collection materialization for $\langle T\,[A]\,\langle E\,[V]\rangle\rangle$.

shows the data materialized in containers and records. As can be seen, the topmost collection $\mathcal{T}$ of type $\langle T\,[A]\,\langle E\,[V]\rangle\rangle$ is materialized in the dedicated container $c(\mathcal{T})$. Similarly, $\mathcal{E}_{Order}$ and $\mathcal{E}_{City}$ are materialized in the dedicated containers $c(\mathcal{E}_{Order})$ and $c(\mathcal{E}_{City})$, respectively. For the remaining collections $\mathcal{A}_{Order}$, $\mathcal{V}_{o_1}$, $\mathcal{V}_{o_2}$, $\mathcal{A}_{City}$, and $\mathcal{V}_{c_2}$ FASE uses embedded materialization. For instance, the collection $\mathcal{A}_{Order}$ completely gets materialized in the record of its nesting group $(Order, \mathcal{A}_{Order}, \mathcal{E}_{Order})$. In the same record, in place of $\mathcal{E}_{Order}$, FASE stores a reference to the dedicated container $c(\mathcal{E}_{Order})$.

4.6.2 Automatic Dislodgment

None of the two materialization strategies is universally superior to the other. While embedded materialization can increase locality and reduce the number of references to resolve, Dedicated materialization avoids very large records and allows selective queries reading only necessary data. Consequently, a fixed decision on which collections should be embedded and which not may lead to suboptimal read performance.

For instance, the embedding strategy used in Figure 4.15 is absolutely reasonable for regular relational data. In typical regular relational data, entities are only a few hundred bytes in size and we have orders of magnitude more entities than attributes. However, with a very large number of attributes or large blob values the embedded collections $\mathcal{A}_{Order}$, $\mathcal{V}_{o_1}$, $\mathcal{V}_{o_2}$, etc. would be significantly larger. If so, a different embedding strategy may be more efficient.

FASE implements an automatic dislodgment algorithm to avoid the embedding of very large collections. With automatic dislodgment, FASE automatically moves collections to dedicated containers once they are getting too large. When a database is created, FASE starts with a single container for the topmost collection. All other collections are materialized in an embedded way. As more data is inserted, the collections grow. Every collection that has embedded materialization size exceeding a configured threshold, e.g. the size of a memory page or a disk block, will be dislodged.

The dislodgment procedure involves four steps. First, FASE creates a new container. Second, it splits the embedded materialization into chunks, one for each group of the collection. Each chunk is split into group header and the other parts of the group (includes and nesting). Third, FASE inserts each chunk as a record into the new container. The group header becomes the key and the other parts of the group become the actual record. Finally, FASE replaces the original embedded materialization with a reference to the new container by updating the original record. The dislodgment happens entirely within in the collection materialization layer of FASE. Since FASE uses the same byte representation of groups in embedded and dedicated materializations, it can directly copy the chunks to records on byte level, minimizing interpretation overhead. This makes the dislodgment procedure very efficient. In any case, the dislodgment procedure is applied automatically only to relatively small collections (just above the threshold), so that operational overhead is small.

Table 4.2: Physical data layouts used in the experiments.

Data layout	Layout definition
Row store	$\langle T\,[A]\,\langle E\,[V]\rangle\rangle$
Interpreted record	$\langle T\,\langle E\,\langle A \rightarrow V\rangle\rangle\rangle$
Column store	$\langle T\,[E]\,\langle A\,[V]\rangle\rangle$
BATs	$\langle T, A\,\langle E \rightarrow V\rangle\rangle$
Bigtable	$\langle T\,\langle E, A \rightarrow V\rangle\rangle$
XML-like	$\langle E \rightarrow T\,\langle A \rightarrow V\rangle\rangle$

4.7 Evaluation

We evaluated FASE in a series of experiments. The focus of FASE is the configurability of the physical data layout. Once configured for a specific data layout, we expect FASE to exhibit similar runtime behavior as a hard-coded implementation of this physical data layout. Note that the aim is not to achieve top-level performance. A specialized implementation targeting a specific field of application can always achieve better performance than a general system built for a broader range of applications. The aim is to increase configurability, physical data independence, and to make specialized physical data layouts available in general systems. In the evaluation we investigated if FASE lives up to this expectation. We start by giving an overview of our prototype and describe the setup of the experiments in Section 4.7.1. In Sections 4.7.2 and 4.7.3, we present the database file size and the load time, respectively. We measured both for the databases that we created with different combinations of data set characteristics and physical data layout. On these databases we ran different workloads and measured their runtime. These results are presented in Section 4.7.4.

4.7.1 Setup

We implemented FASE in Java and ran our experiments with the Java 7 Update 5, 64 bit VM on a machine with an AMD Opteron CPU at 2.6 GHz, 512 KB L2 cache, DDR II memory, and a SATA disk. The prototype is strictly single threaded. All specifiers, i.e. entity types, entity ids, and attributes, are dictionary-encoded as 8-byte integers. Record containers are implemented as disk-based B-trees. In a container, the B-trees indexes the keys under which the records are stored. For unordered collections, the group header becomes the key. For ordered collections, the order (as 4-byte integer) becomes the key. Besides these container B-trees, there are no other indexes involved. The B-trees use pages 8 KB in size and a shared buffer of 8192 pages, i.e. 64 MB in size. This disk-oriented setup mimics the situation of traditional general purpose database systems. Nevertheless, disk-orientation is not essential for the concepts of FASE, and neither are B-trees. A FASE-based data layout could also be implemented for main memory and with other data structures, or with a mixture of data structures.

Table 4.3: Data sets used in the experiments.

Data set	Number of			Sparse	Raw size
	Entity types	Entities per type	Attributes		
256K-entities	1	262 144	16	1.0	64 MB
512K-entities	1	524 288	16	1.0	128 MB
1024K-entities	1	1 048 576	16	1.0	256 MB
0%-sparse	1	1 048 576	16	1.0	256 MB
50%-sparse	1	1 048 576	32	0.5	256 MB
80%-sparse	1	1 048 576	80	0.2	256 MB
512-types	512	2048	16	1.0	256 MB
1024-types	1024	1024	16	1.0	256 MB
2048-types	2048	512	16	1.0	256 MB

Table 4.4: Workloads used in the experiments.

Data set	Number of queries	Query pattern
row-oriented	100	$(\textsc{Rnd}, \textsc{Rnd}, *)$
column-oriented	1	$(\textsc{Rnd}, *, \textsc{Rnd})$
entity-oriented	number of entities per type	$(*, \textsc{Rnd}, *)$
entity type-oriented	1	$(\textsc{Rnd}, *, *)$

In multiple experiments, we configured FASE with the FASE expressions shown in Table 4.2. In each configuration, we load the different data sets and measured the load time as well as the resulting database file size. For the resulting databases, we measured the execution time of different workloads.

To obtain data with specific properties, we generated nine synthetic data sets. Each data set consists of a given set of entity types, each with the same number of entities. From a given set of attributes, each entity instantiates an individually and randomly chosen subset. The size of the subset depends on a given fill factor, e.g. for a fill factor of 0.5 half of the attributes are instantiated, while for a fill factor of 1.0 all attributes are instantiated. The fill factor allows creating sparse data sets. Table 4.3 lists the nine data sets we generated with their respective parameters. All values are randomly generated 16-byte character strings, so that the resulting raw data size (values only) is as shown in the table. Note that the data sets 1024K-entities and 0%-sparse are equal. We distinquish these two solely for presentation purposes.

We used four synthetic workloads in total. All workloads consist of a given number of randomly generated queries, i.e. calls of the Select operation of the logical data interface of FASE (cf. Section 4.4). The generated qualifier triple for each Select call contains at each position either a randomly selected specifier or the wildcard $*$

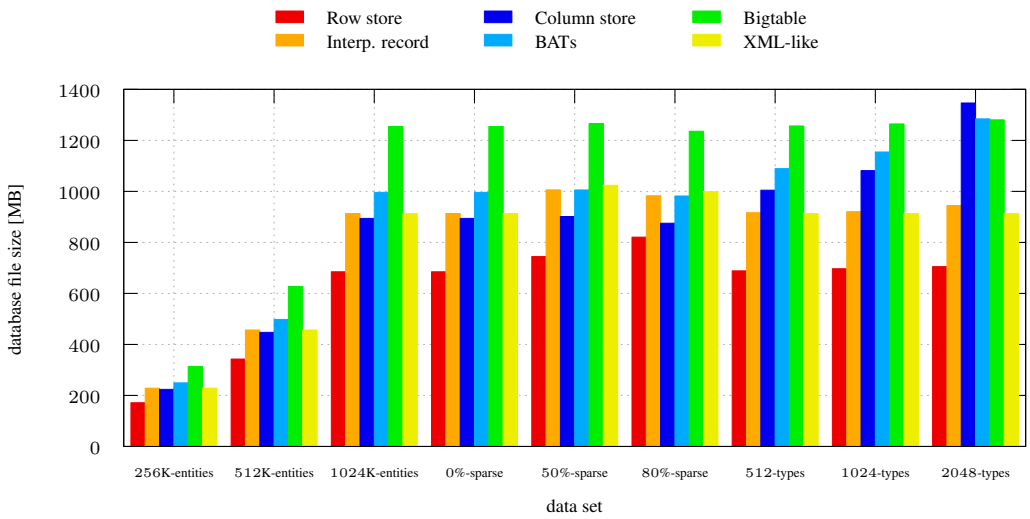


Figure 4.16: Database file size for different data sets and physical data layouts in FASE.

depending on the query pattern given for the workload. Table 4.4 lists the four workloads with the number of queries each workload encompasses and the workload's query pattern. In the query patterns, RND indicates a randomly selected specifier.

4.7.2 Database File Size

To investigate the influence of the physical data layout on the database file size, we measured the database file size for all data sets and all physical data layouts after the data had been loaded. Figure 4.16 shows the database file size in MB for each combination of data set and physical data layout. As can be seen, the physical data layout has a notable influence on the database file size. In general, the row store layout is the most space-efficient one, followed by interpreted record, column store, BATs, and XML-like with medium space efficiency and Bigtable as the least space-efficient layout. Row store is the most space-efficient layout because it uses order to relate values to specifiers, and thereby avoids redundancy. All layouts without ordered collections, namely interpreted record, BATs, XML-like, and Bigtable have to store specifiers multiple times. Interpreted record has redundancy on the attribute specifiers, BATs on the entity specifiers, XML-like on the entity type specifiers and attribute specifiers, and Bigtable on the entity specifiers and attribute specifiers.

Judging from the specifier redundancy, the column store layout and the Bigtable layout should be at least as space-efficient as the row store layout and as the XML-like layout, respectively. However, this is not the case because our prototype solely uses B-Trees as container implementation. On the column store layout, automatic dislodgment results in dedicated materialization of the ordered collections, in contrast to the row store layout. The order is the key of ordered collections, so that dedicated

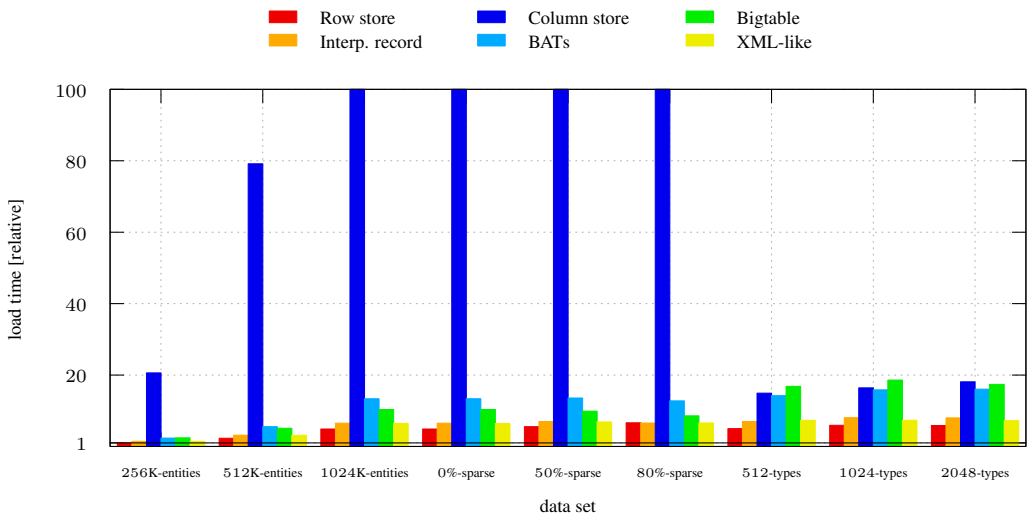


Figure 4.17: Load time for different data set and physical data layouts.

materialization of ordered collections stores the order as integer in the B-tree key, which reduces the space efficiency. In a more mature implementation, one would use a different data structure for ordered collections, which would store the order implicitly. The Bigtable layout stores all values belonging to the same entity type in a single collection with a composite header (E and A). In a B-tree, this translates into keys twice as long as the keys of collections with a header consisting of a single specifier. Larger keys reduce the fanout of a B-tree and lower its space efficiency.

The obvious exception from the general picture are the data sets 512-types, 1024-types, and 2048-types and here in particular the column-oriented data layouts column store and BATs. In both layouts, the number of entity types and the number of entities per type have considerable influence. Note that the total number of entities and the total number of values is the same in all three data sets. The more entity types are in the database, the higher the number of collections that hold the values and the smaller these collections are. Basically, a larger number of entity types results in a finer partitioning of the values. With the column-oriented layouts, these collections are materialized as B-trees. A B-tree needs additional space for the directory pages. The smaller the B-tree is, the larger its share of directory pages gets. Consequently, the space efficiency of the column-oriented layouts is inversely proportional to the number of entity types for a fixed number of entities.

4.7.3 Load Time

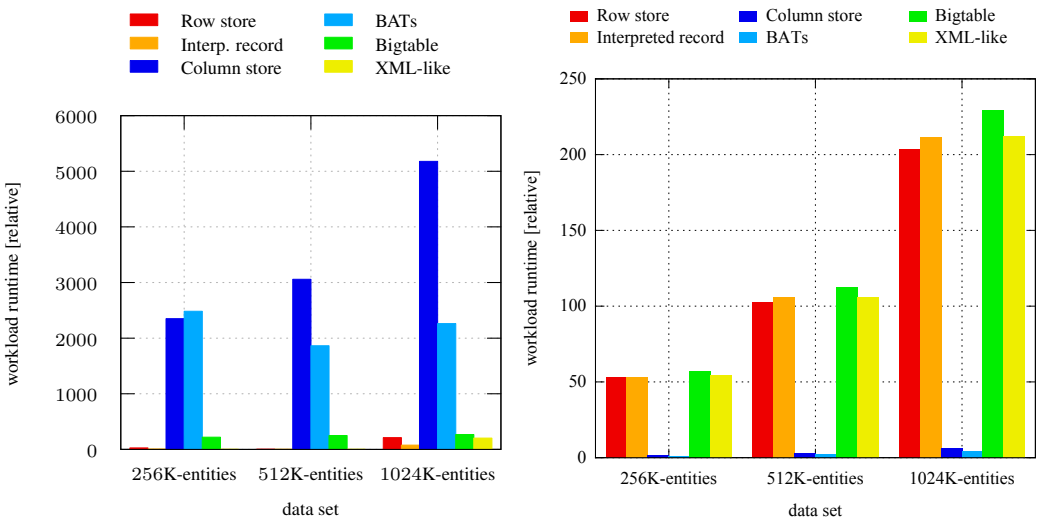
We also measured the load time for all data sets and all physical data layouts. The data sets were loaded with the INSERT operation of the logical data interface of FASE (cf. Section 4.4). Each INSERT call takes 16 quadruples. For the column store layout and the BATs layout, quadruples were grouped attribute-wise and ordered

within a group according to the entity. For the other layouts, quadruples were grouped entity-wise and ordered within a group according to the attribute. Of course, this is an idealized situation and differs from a normal write-intensive workload, where quadruples would show up in a more randomized fashion. Nevertheless, the load time allows drawing conclusions about the general write performance of the different layouts. Figure 4.16 shows the load time relative to the 256K-entities data set with the row store layout. Generally, we see that the load time follows the database file size. When we have more raw data or where the data layout requires more bytes to be stored the load time is higher.

The main exception from this general picture is the column store layout. To insert a single value in the column store layout, FASE has to find the matching entity in the $[E]$ collection of the corresponding entity type to determine the position of the value in its $[V]$ collection. Since $[E]$ collections are typically stored in a dedicated container, our prototype scans the leaves of the corresponding B-tree to find the matching entity. This imposes a considerable cost on the insert operation if the column store layout is used. The cost is even higher than the data is sparse. Our implementation cancels the scan over the entities as soon as it has found matches for all quadruples that have to be inserted. On dense data, values for each attribute are inserted in the same order regarding their entity, so that $[E]$ collection also has that order and the scan is canceled early for most of the insert calls. On sparse data, by contrast, entities that do not instantiate the first attributes end up further down in the entity collection than they appear in the inserts of the attributes that these entities have instantiated. As a result, there are fewer opportunities to cancel the scan early. This dramatically illustrates the fact that the column store layout is not a write-optimized layout. In a more mature implementation, however, entity specifiers could be assigned not by the user but by the database system. In this case, the order itself can be used as an implicit entity specifier. Implicit entity specifiers remove the necessity of explicitly storing and scanning $[E]$ collections and significantly improve the runtime performance of the INSERT operation on the column store layout.

4.7.4 Workload Runtime

As is well-known from hard-coded implementations of physical data layouts, certain workloads and data set characteristics favor certain physical data layouts. To investigate if FASE is able to reflect this, we measured the runtime of different kinds of workloads on different data sets and different physical data layouts. FASE should exhibit a considerably lower workload runtime for the favorable combinations. We show the workload runtime relative to the lowest runtime measured for a given workload among the considered settings. In other words, each plot shows how much longer the displayed workload took to be processed compared to the fastest combination of data set and physical data layout. Specifically, we investigated two pairs of workloads: (1) the row-oriented workload vs. the column-oriented workload, and (2) the entity-oriented workload vs. the entity type-oriented workload. The results are presented in the following for each pair separately.



(a) row-oriented.

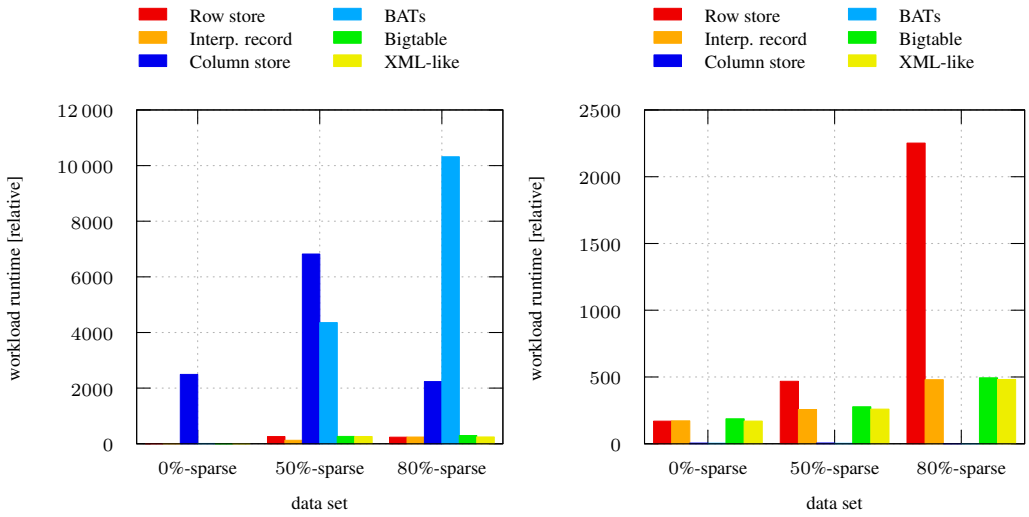
(b) column-oriented.

Figure 4.18: Row-oriented and column-oriented workloads on regular data.

Row-oriented vs. Column-oriented Workloads

We measured the row-oriented workload and the column-oriented workload (cf. Table 4.4) first on the regular data sets 256K-entities, 512K-entities, and 1024K-entities, and second on the irregular data sets 0%-sparse, 50%-sparse, and 80%-sparse (cf. Table 4.3). The row-oriented workload retrieves randomly chosen entities of a given entity type, while the column-oriented workload retrieves all values of a randomly chosen attribute of a given entity type. Hence, the row-oriented workload favors physical data layouts that (1) cluster all values of an individual entity and (2) allow fast lookups of an entity type specifier and an entity specifier. In contrast, the column-oriented workload favors physical data layouts that (1) cluster all values of the same attribute and the same entity type, and (2) allow fast lookups of an entity type specifier and an attribute specifier.

Figure 4.18 shows the results for the regular data sets; Figure 4.18(a) for the row-oriented workload and Figure 4.18(b) for the column-oriented workload. The row-oriented workload is significantly slowed down by the column-oriented layouts, and in particular by the column store layout. The column store layout is the only of the tested layouts that neither groups values per entity, nor provides a fast entity lookup. Consequently, the row-oriented workload needs two orders of magnitude longer to be processed on column store layout than on any other layout. Note that this difference is mainly caused by scans of $[E]$ collections necessary to lookup an entity specifier. Again, implicit entity specifiers would significantly improve the performance here (cf. Section 4.7.3). The column-oriented workload shows the opposite picture. Here, all row-oriented layouts (row store, interpreted record, Bigtable, and XML-like)



(a) row-oriented.

(b) column-oriented.

Figure 4.19: Row-oriented and column-oriented workloads on irregular data.

show a one order of magnitude longer workload runtime than the column-oriented layouts (column store and BATs). This is as expected, as these layouts do not cluster values by attribute and require FASE to read over the complete data set to retrieve the values that belong to a single attribute.

Figure 4.19 shows the results for the irregular data sets; Figure 4.19(a) for the row-oriented workload and Figure 4.19(b) for the column-oriented workload. On irregular data, we see the same general patter as on regular data; a row-oriented workload benefits from a row-oriented layout and a column-oriented workload benefits from a column-oriented workload. However, the sparseness of the data affects the different layouts differently. As all three irregular data sets have the same raw data size, the same number of entities, and the same average number of attributes per entity, the total number of attributes increases to allow for sparser data sets. This explains the visible effects on the workload runtime.

With the row-oriented workloads, the negative effect of the column-oriented layouts gets worse with increasing sparseness. With every additional attribute, FASE has to check an additional B-tree if it contains a value of the entities to retrieve. At the same time, however, the number of values per attribute decreases with increasing sparseness. If the number of values is small enough so that the B-tree requires fewer levels of directory nodes, the cost of an individual B-tree seek decreases, too. As we can see for 80%-sparse data set with the column store layout, this decrease of individual seek cost can compensate the increase in the number of required seeks. We do not see this effect for the BATs layout because the BATs layout uses the 8-byte long entity specifier as key in the value B-trees, while the column store layout has the only 4-byte long order as B-tree key. The larger keys reduce the fanout of the

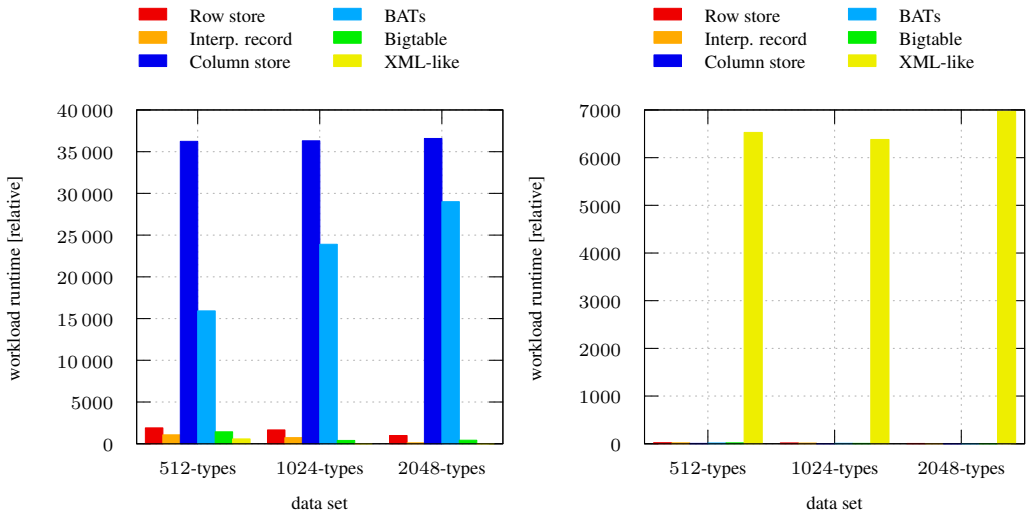
B-trees in the BATs layout so that 80%-sparse data set is not sparse enough to allow the compensation effect.

With the column-oriented workload, the performance of the row store layout is negatively affected by increasing sparseness more than the other row-oriented layouts. The row store layout has to represents missing attributes explicitly for each entity. Although, this does not require much additional space compared to the size of a value, it increases the size of the entities enough to reduce blocking factor of the leave pages of the B-tree that stores the $\langle E\,[V]\rangle$ collection. As a consequence, FASE has to scan more pages read all entities. Additionally, the larger number of attributes increases the interpretation overhead for each entity.

Entity-oriented vs. Entity Type-oriented Workloads

We measured the entity-oriented workload and the entity type-oriented workload (cf. Table 4.4) on the data sets 512-types, 1024-types, and 2048-types (cf. Table 4.3). The entity-oriented workload retrieves entities regardless of their entity type, while the entity type-oriented workload retrieves all entities of a given entity type. To be comparable in the amount of data both workloads retrieve, the entity-oriented workload queries as many randomly chosen entities as there are entities per type. For instance, in the data set 512-types, both workloads retrieve 2048 entities in total. The three data sets differ in their entity-type–entity ratio, but have the same total number of entities and the same raw data size. For the entity-oriented workload, physical data layouts are expected to be beneficial that (1) cluster all values of an individual entity, and (2) that allow fast lookups of an entity specifier. In contrast for the entity type-oriented workload, physical data layouts are expected to be beneficial that (1) cluster all entities of an individual entity type, and (2) that allow fast lookups of an entity type specifier.

The results are shown in Figure 4.20; Figure 4.20(a) for the entity-oriented workload and Figure 4.20(b) for the entity type-oriented workload. Similar to the results of the entity type-oriented workload, the column store layout shows the worst performance on the entity-oriented workload. Again, the reason for the drastic difference is the necessary scan over $[E]$ collections to find an entity. Naturally, the workload runtime on the column store layout gets better the fewer entities per entity type the data set has. The entity type-oriented workload shows the opposite picture. Here, the purely entity-oriented XML-like layout considerably slows the workload down. To find all entities of a given type on the XML-like layout, FASE has to read all entities. With more entity types and fewer entities per entity type in the data set, less data has to be retrieved by the entity type-oriented workload. While the workload runtime on all other layouts decreases with the number of entities per entity type, it remains the same for the XML-like layout. This underlines that the XML-like layout is simply not designed for type-oriented retrievals.



(a) entity-oriented.

(b) entity type-oriented.

Figure 4.20: Entity-oriented and entity-type-oriented workloads.

4.8 Summary

There is not a single optimal physical data layout for the irregular, self-descriptive data of schema-comes-second data management. FASE is a storage engine that can be configured to various data layouts using the FASE notation. The FASE notation allows describing of the macroscopic characteristics of physical data layouts, i.e. how data elements are grouped in the physical storage. More specifically, a FASE expression describes a hierarchical structure of nested collections, which contain groups of data elements and thereby determine how data elements can be selected and scanned. The FASE notation can express common physical data layouts, such as row store, column store, BATs as well as data layouts such as interpreted record or vertical schema, which were designed for irregular data. Beside the FASE notation, we discussed the principal architecture of FASE and how it implements fundamental CRUD operations. In the evaluation, we showed that FASE exhibits the performance characteristics of the physical data layout it was configured for. FASE is not meant to provide top-level performance but to allow combining the characteristics of various physical data layouts in a single system. Although our prototype is far away from providing productivity-ready performance, it allows demonstrating the benefits of the FASE concept. In the evaluation, we have seen that FASE is able to provide the benefit certain layouts have for certain workloads. We have not evaluated the overhead that has to be paid for the configurability of FASE. Such an evaluation requires a more mature implementation, which is more comparable to hard-coded implementations of different physical data layouts than our prototype is. Undeniably, the configurability of FASE comes to the price of additional interpretation overhead. However, code

generation techniques such as micro specialization [Zhang et al., 2012a,b] can help to remove interpretation overhead from a database system. A combination of FASE and micro specialization can retain the flexibility of FASE but achieve performance at the level of hard-coded implementations. As we have also seen in the evaluation, the configurability of FASE is particularly useful with the irregularly structured, self-descriptive data of a FRDM database, as there is no single physical data layout optimal for every data set and every workload.

5 SMIX – Self-Managing Indexes

> B-trees are by far the most important access path structure in database and file systems.
>
> *Jim Gray and Andreas Reuter [Gray and Reuter, 1993]*

Indexes are the most fundamental technique to speed up queries in database systems. Based on the usage, and independent of the specific structure of an index, we can distinquish two basic types of indexes. Primary indexes are not part of the primary store that organizes the payload data as we have seen in Chapter 4. Secondary indexes, in contrast, are auxiliary structures that store additional information about the physical location of data to facilitate fast access to this data. A typical secondary index redundantly stores values of the indexed attribute. The internal structure of an index allows quick finding of values that match a given query predicate. Each value is associated with the identifiers of the entities that instantiate the indexed attribute with that value. If other values of the matching entities need to be retrieved, they have to be fetched from the primary store.

In the FASE termininology of Chapter 4, we can describe a secondary index as an additional collection of type $\langle V \rightarrow E \rangle$, which exists in addition to and in sync with the primary store. The totality of all indexes created in a database is often called the index configuration of the database. Simplifying, we can denote an index configuration as $\langle A' \langle V \rightarrow E \rangle \rangle$, where A' is a subset of A. Creating an index configuration essentially requires determining A'. Since each individual secondary index supports only a fraction of the database's workload while requiring resources to be stored and maintained, creating an index configuration is an optimization problem. With changing data and shifting workloads, the optimum is a moving target, though. As a secondary data structure, indexes always constitute a trade-off between increased query performance on the one hand and storage resources and maintenance cost on the other hand. Index information that is beneficial today may be unprofitable tomorrow, while another index may have become very useful at the same time. Index optimization is not a decision made at a single point in time, but remains a continuous effort. Self-managed indexing, where index optimization is an integral part of the database system, can relieve the DBA permanently of this burden.

Also schema-comes-second databases have to rely on secondary indexing for efficient query processing. In schema-comes-second databases, however, determining A' manually becomes even more complicated since A is not fixed and not necessarily known in advance. In other words, with the schema being practically unknown at design time, it is hard to create a reasonable index configuration in the first place. A schema constantly in flux compromises the effectiveness of a traditional index

configuration over a longer time period. Each schema change by the database user requires the DBA to retune the index configuration. Hence, self-managed indexing without the DBA in the loop is an essential requirement for schema-comes-second database management systems.

Self-Managing Indexes (SMIX) are a novel, adaptive, fine-grained, autonomous indexing infrastructure. It builds on a novel access path that automatically collects useful index information, discards useless index information, and competes with its kind for resources. In contrast to existing technologies for adaptive indexing, the SMIX concept is able to grow and shrink indexes dynamically, instead of incrementally enhancing the index granularity. With its autonomous adaptability, SMIX is suitable for databases with very dynamic workloads as well as schema-comes-second databases with dynamic schemas.

In the Section 5.1, we review existing index tuning and adaptation techniques. In Section 5.2, we give an overview of the SMIX concept, followed by more detailed discussions in Sections 5.3 and 5.4. Section 5.5 presents an extension to the core idea of SMIX. Then, we elaborate evaluation results in Section 5.6 und summarize the chapter in Section 5.7.

5.1 Related Work

Substantial research has been done in the field of automatic index tuning. First research in this area dates back to the late 1970s. Nowadays commercial database management systems offer index tuning tools, which recommend an index configuration for a given workload and storage bound the configuration has to fit into. Since the configurations recommended by all of these state-of-the-art tools do not consider that database workloads may change over time, they can be called *static* configurations. There are also *dynamic* configurations, i.e. configurations that adapt the chosen set of index structures over time. Research has addressed dynamic index recommendations in the last decade. Static as well as dynamic index configurations consider only full-column indexes completely created or dropped at a time. In more recent years, research index tuning turned to *partial* indexing. Here, the database system does not have a configured set of indexes, but a subset of tuples indexed in each column.

It is also possible to distinguish between *on-line*, *off-line*, and *inherent* index tuning techniques. On-line tools continuously monitor the workload. They base their decisions on the knowledge gained from the recently observed workload. By contrast, off-line tools do not monitor the workload. Their input may be a monitored workload, but it may be a manually constructed workload, too. Both on-line and off-line share the property that the index tuning is implemented in a dedicated tool or component, which exists separately from the query processing engine. With inherent index tuning, indexing becomes an integral part of the query processing and index tuning dissolve as a separate entity. The workload is not monitored to feed some configuration component, but queries actively build indexes while they are processed by the database system.

	off-line (tool)	on-line (tool)	inherent
static (complete)	*advisor*	*alerter*	-
dynamic (complete)	*dynamic advisor*	*auto-tuning*	-
partial	-	*partial tuning*	*adaptive indexing*

Table 5.1: Classification of index tuning techniques.

Static/dynamic/partial and *on-line/off-line/inherent* are two dimensions of orthogonal sense. Together, both form a meaningful classification of automatic index tuning techniques as shown in Table 5.1.

Today's commercial index advisor tools realize *static, off-line* tuning. The DBA runs these tools off-line and obtains an index configuration. A advisor is unquestionably of great help. But the DBA still has a significant share of work to do to tune the system and keep it tuned. He has to run the advisor, monitor the system constantly and eventually run the advisor again if the index configuration turns out to be no longer optimal. At this point, an alerter is the tool of choice. It is a *static, on-line* tool, constantly checking whether the current index configuration is still acceptable and alerting the DBA if not. By putting the functionality of both advisor and alerter together, we obtain auto-tuning. An auto-tuning facility constantly checks the current index configuration and changes it to a more adequate one if necessary. It therefore belongs to the class of *dynamic, on-line* techniques. Auto-tuning is appealing because it attempts to automatically adjust the index configuration to account for changes in the workload over time. However, being an on-line mechanism, it can consider only the workload that it has already observed and has to predict the future based on this observation. Where the workload changes over time but in a well-defined manner, a *dynamic, off-line* tuning tool achieves better results. Such a dynamic advisor gets the workload as a sequence of database statements (rather than as a set of statements) and recommends a sequence of index configurations. If the workload is not well-defined, though, one has to live with auto-tuning. Traditional auto-tuning creates and drops complete indexes, i.e. configuration changes and wrong tuning decisions are costly. A *partial, on-line* tuning tool tries to reduce the cost by making tuning decisions on a tuple basis rather than on a column basis. Each tuning decision, then, causes only small incremental changes to the index configuration. As a positive side effect, partial indexing avoids spending indexing resources on tuples that are never queried with an index. The small incremental changes to what is indexed have such a low cost that it is even possible to directly integrate them into the query processing. The resulting *partial, inherent* index tuning is often referred to as adaptive indexing. SMIX is in the class of adaptive indexing.

In the following, we present an overview of the work existing in each of the six classes of index tuning.

5.1.1 Advisor

Research in the area of physical design in general and index tuning specifically started early and already included the question of how certain tuning decisions can be made automatically. A survey of the early research in the field of physical design tuning can be found in [Schkolnick, 1978]. With the relational database model [Codd, 1970], the B*-Tree [Comer, 1979], and relational database systems setting out to conquer the database world the question of an adequate index configuration for a given workload became an essential part of the lives of DBAs. Later, when large distributed database systems and data warehouse systems became widely popular, materialized views and partitioning became an important part, too.

Early on, Comer [1978] proved optimal index tuning to be an NP-complete problem; even in a very restricted setup. The complexity of the problem arises because every subset of the set P of possible indexes is a potential solution. Each of those 2^P possible solutions needs to be checked to find the optimal one.

While much research focused on appropriate decision models particularly for secondary index selection [Ip et al., 1983, Bonnano et al., 1985, Hatzopoulos and Kollias, 1985], Finkelstein et al. [1988] were the first to actually build an index selection tool. To be in aligned with the decision model of the database system they developed a technique nowadays well known as *what-if* utility [Chaudhuri and Narasayya, 1998]. This utility takes a configuration and a query. It manipulates the system catalog to simulate the configuration and asks the optimizer for the estimated cost of the query. Afterwards, it cleans up the system catalog again and returns the cost of the query. The what-if utility is capable of determining the estimated execution cost of a query under a certain index configuration without creating the indexes.

However, what-if calls are expensive, so minimizing their number is an essential goal. Unfortunately, in the presence of common join methods, such as nested loop and sort merge, we cannot judge every index separately. Finkelstein et al. solved this with the idea of *atomic configurations*. Atomic configurations have only one index per table. If a query execution plan uses exactly one access path for each appearance of a table in the query, the cost of the query using a configuration that is assembled from atomic configurations can be derived from the query's cost using atomic configurations individually.

Although atomic configurations reduce the number of what-if calls, they do not reduce the search space. Finkelstein et al. used two techniques to reduce the search space. First, the search space can be reduced constructively; instead of generating all possible atomic configurations, we consider only columns that are plausible for indexing. For instance, a column that is not referenced by a query at all is implausible to be indexed for that query. Second, the search space can be reduced deconstructively; if we know how much cost the atomic configurations save for which statement, we can use heuristics to prune indexes of low value for the complete workload.

In the global picture Finkelstein's index tuning tool, which later became IBM's Relational Design Tool, looks as follows. The tool begins by scanning the workload. Thereafter, it finds the referenced tables and columns plausible for indexing for

each statement, creates all atomic configurations of indexes on plausible columns and determines the statement's execution cost under these configurations. Finally, it removes heuristically less cost-saving indexes from the set of candidates and enumerates the remaining search space to find the optimal configuration.

All the concepts described in the previous paragraphs – atomic configuration, what-if interface, constructive reduction, destructive reduction and enumerating candidates – can be found in modern design tools. Of course, the tools differ in how the concepts are implemented and put together. Nevertheless, those concepts are the fundamental building blocks of today's advisors.

Incrementally, steps were made toward more general design advisors. The first index selection tool for Microsoft's SQL Server also considers multi-column indexes Chaudhuri and Narasayya [1997]. Later it was extended to materialized views [Agrawal et al., 2000]. By adding partitioning [Agrawal et al., 2004] Microsoft completed its Database Tuning Advisor. In the meantime IBM redeveloped their index advisor tool for DB2, bound it tighter to the optimizer [Valentin et al., 2000] and extended it to a general design tool [Rao et al., 2002, Lightstone and Bhattacharjee, 2004, Zilio et al., 2004]. Dageville et al. [2004] did the same for Oracle.

All state-of-the-art index advisors work well. Nevertheless, researchers stayed with the topic and examined new approaches. Two are worth mentioning.

Chaudhuri and colleagues defined the operations *merge* and *reduce* to combine and transform, respectively, indexes to other indexes. Both operations work with materialized views, too. The resulting index of each of the two operations needs equal or less space than the input does. *Merge* and *reduce* allow refining an optimal configuration that violates the space constraint step by step to smaller configurations until the space constraint is met. A smaller configuration is usually less good than a larger one. Each transformation is a trade-off between space saving and query performance. The whole process is called *design refinement* or *design relaxation* [Chaudhuri and Narasayya, 1999, Bruno and Chaudhuri, 2005, 2006a, 2007b].

IBM showed that it is worth binding the design tool tighter to the optimizer [Valentin et al., 2000]. Bruno and Chaudhuri drove this idea a bit further. They implemented a new interface to the optimizer, which allows extracting of which single-table requests the optimizer considers. Because there is a limited number of possible access-paths for a given single-table request, they are able to simulate all possible access-paths for that request in the system catalog and let the optimizer decide which is the most cost-effective one. With this they are able to determine the best overall configuration – without regarding any space bound – for one query with only one optimizer call. Combined with a design refinement algorithm, this provides a new approach to building an index advisor [Bruno and Chaudhuri, 2005]. Further work also aims at binding the advisor tighter to the optimizer and reduce the number of what-if round-trips [Papadomanolakis et al., 2007, Bruno and Nehme, 2008].

There is a number of other publications about index selection, which we will not discuss in further detail, e.g. [Choenni et al., 1993, Caprara et al., 1995, Gupta et al., 1997, Chaudhuri et al., 2004].

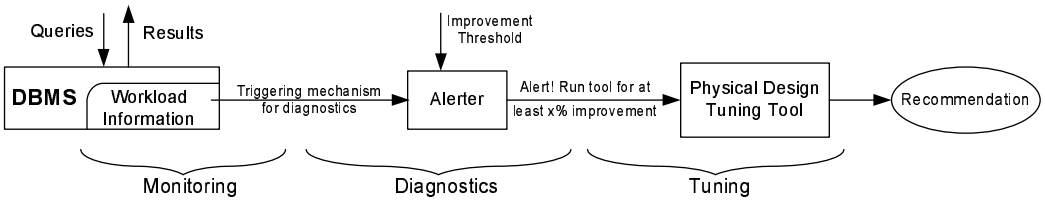


Figure 5.1: Usage of the design alerter. *Source:* [Bruno and Chaudhuri, 2006b]

5.1.2 Alerter

Index advisors are a good choice if the database workload and the data in the database are rather static. However, the workload or the data may change from time to time. The DBA needs to know at which point in time he has to retune the system. Since index advisors make many optimizer calls during one tuning session, they cause a significant load on the database system. Thus, the advisor is not the right to tell the DBA whether he has to retune the system or not. Although auto-tuning would run continuously, it is also not ideal for workloads that change infrequently. Most of the time, an auto-tuning tool would consume computing power only to find out that nothing has changed. In such situations an alerter is the tool of choice.

Alerters have only scarcely been considered so far. Only Bruno and Chaudhuri [2006b] proposed a solution. Figure 5.1 shows how such an alerter basically works. The alerter monitors the database and runs its diagnostics periodically, for example triggered by a scheduled job. In case the index configuration is unprofitable it posts an alert message, so that a new tuning session can take place. Bruno and Chaudhuri used their optimizer instrumentalization (cf. page 101) to implement their design alerter. With this instrumentalization of the query optimizer, they are able to log detailed information about the workload and how the optimizer deals with it. In particular, they log which alternate single-table access paths the optimizer has to satisfy a single-table access, and the cost of these access paths. By modifying the workload's access plans with these paths locally, the alerter can find out whether a more profitable configuration exists. It is a low-overhead procedure that avoids additional optimizer calls.

5.1.3 Auto-Tuning

As explained before, an alerter is a useful tool to inform the DBA when a usually static database system needs a new index configuration. Obviously, alerters become impractical if the workload or the data or both are highly dynamic. Assume that a database requires a new index configuration once a day on average. Under such circumstances, being heavily involved in the index tuning is undesirable for the DBA. This would devour too much of valuable working time. What is needed is auto-tuning of the index configuration.

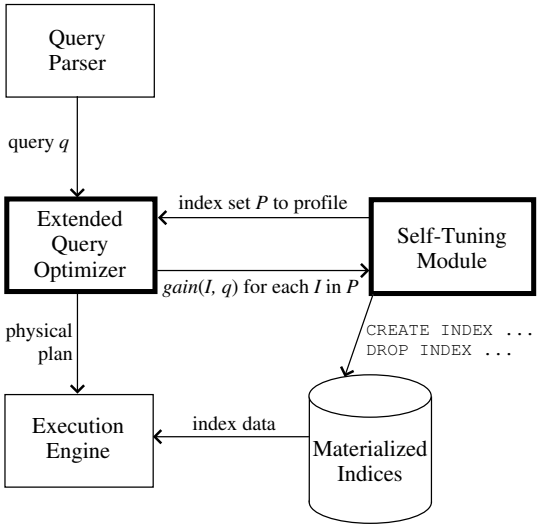


Figure 5.2: COLT architecture. *Source:* [Schnaitter et al., 2006]

Hammer and Chan [1976] considered auto-tuning ahead of time. At the end of the 1970s the database world was not ready for the topic yet. Researchers became highly interested in index auto-tuning two and a half decades later when more and more IT vendors hopped on IBM's *autonomous computing* wagon [Kephart and Chess, 2003, IBM, 2001]. Several researcher groups worked on the topic. All auto-tuning approaches have in common that they build on an *Observation-Prediction-Reaction*-cycle [Weikum et al., 1994]. We present three of these approaches in the following paragraphs.

COLT (Continuous On-Line Tuning) is "a self-tuning framework that continuously monitors the incoming queries and adjusts the database design in order to maximize query performance" [Schnaitter et al., 2006, 2007]. It uses a modified query optimizer as depicted in Figure 5.2 as *Extended Query Optimizer*. The extended optimizer evaluates the cost saving by materialized and hypothetical indexes for the current query. In other words, it implements an *on-line what-if* interface. COLT's other essential component is the *Self-Tuning Module*. The Self-Tuning Module does the actual tuning iteratively. In each round it profiles index candidates and maintains the current configuration. First, it profiles a subset of all candidate indexes for a certain number of queries by using the on-line what-if interface. Afterwards, the Self-Tuning Module decides which of the indexes it profiles in the next round and if the configuration has to be changed. It triggers the changes if necessary. The Self-Tuning Module's decisions are based on cost the materialized and the hypothetical indexes saved or would have saved, respectively. Most of COLT's processing overhead arises when it calls the on-line what-if interface. The Self-Tuning Module is able to lower its overhead by adjusting the number of indexes that it profiles per query. It profiles only the most promising indexes. If the system is well-tuned the number of promising

indexes is low. The fewer indexes the Self-Tuning Module profiles, the less overhead it generates. When the workload shifts and retuning becomes necessary, the number of promising indexes grows and COLT gets more aggressive.

Sattler's index auto-tuning approach [Sattler et al., 2003, 2004, 2007] is quite similar to COLT. The main tuning component works iteratively, too. It watches the queries coming in and determines the potential cost saving by several candidate indexes. At the end of each round the component decides whether it has to change the index configuration or not. Its decisions are based on a cost model and heuristics. Sattler's work differs from COLT in two essential points. The first difference concerns the on-line what-if interface. Sattler and colleagues did not modify the optimizer for cost determination. Instead, they call the optimizer twice for each query, once without considering any indexes, and then considering all index candidates. This does not allow them to adapt the tuning overhead to the dynamic of the workload, as COLT can do. The second difference is in regards to index creation. Sattler and colleagues adapted the idea of Graefe [2000] and integrated index creation with the query processing. Therefore, they implemented two new plan operators. The new operators allow a query that misses a useful index to create this index while it reads the data. As a result, the data is read only once, the query has the index present for its next execution and index creation cost is significantly lower.

Bruno and Chaudhuri [2007a] developed an auto-tuning facility based on their design alerter (see Section 5.1.2). Like the alerter, the auto-tuning facility uses their optimizer instrumentalization and local plan transformation to evaluate alternate single-table access-paths for incoming queries without additional optimizer calls. Based on this evaluation they decide whether the index configuration needs to be changed or not. Furthermore, Bruno and Chaudhuri propose to suspend an index instead of deleting it. A suspended index is not updated and cannot help query processing, although, it occupies storage. The later restart of the index is done by propagating changes from the log to the index. Bruno and Chaudhuri claim this procedure to be generally faster than a full rebuild.

5.1.4 Dynamic Advisor

Auto-tuning (Section 5.1.4) and alerters (Section 5.1.2) are an appropriate choice if the database workload fluctuates unevenly and does not follow any regular pattern. However, if workload changes follow a known pattern on-line techniques are unnecessary for two reasons: (1) They put an extra load on the database system only to rediscover what is already known. (2) They cannot predict a workload shift; they can only react to the shift. Thus, if the DBA knows about patterns in the workload, a dynamic advisor is desirable.

The idea of a dynamic advisor, in contrast to a normal index advisor, is to exploit the ordering of the statements in the given workload and to recommend a sequence of index configurations that reflects the fluctuations in the workload. Agrawal et al. [2006] were the first to propose such a dynamic advisor [Agrawal et al., 2006]. In Agrawal's approach the recommended index configuration sequence assigns a

particular configuration to each statement in the input workload. In other words, the recommendation fits a certain sequence of statements. This works fine if an application only runs a predetermined set of queries in a predetermined order against the database system, for instance a dedicated data mart that automatically generates analytic reports on a daily basis. In situations like this, we can tailor the index configuration to the needs of the predetermined query sequence of the application. The recommended configuration sequence might indicate that a particular index should be created prior to the execution of a specific query in the sequence, and then dropped in favor of a different index for a subsequent query.

However, other common applications, such as online shops or any kind of multi-user applications with concurrent transactions do not have an exactly predetermined sequence of statements that they will run against the database. For this type of application we do not know in advance which sequence of statements the system will be faced with. Consequently, a workload is a sample rather than an exact representation. The productive workload is expected to show deviations from the sample. This fact can be taken into account to recommend a suitable configuration sequence by restricting the number of configuration changes [Voigt et al., 2008]. Still it is up to the optimization algorithm to find out where the changes should be made to minimize the workload execution cost.

5.1.5 Partial Indexing

Partial indexing is a concept first introduced in the late 1980s [Stonebraker, 1989]. Instead of indexing all values in a column, a partial index covers a subset of values given by a predicate. For a given query the database system can check if the query predicate is covered by the predicate that defines the partial index. In case the partial index can answer the query, it is up to the DBA to decide which partial indexes are useful. The original idea is to create partial indexes along the query predicates that can be found in the workload. Nevertheless, database optimizers typically decide to use an index only if the query predicate is selective enough. Consequently, it is reasonable to use partial indexing also to avoid the indexation of values that will never be accessed with the help of an index [Seshadri and Swami, 1995]. To be able also to take this into consideration, a DBA or a tuning tool has to look at the workload and at the data distribution. Seshadri and Swami outlined such a predicate selection procedure for partial indexing. Apart from that, partial indexes did not gain much attention in the field of automatic index tuning.

More recently, however, Wu and Madden [2011] proposed an auto-tuned partial indexing tool called *Shinobi*. In essence, the idea is to range partition tables into interesting tuples and uninteresting tuples and index only the partition of interesting tuples. Like an auto-tuning component, Shinobi continuously monitors the workload of a database system. As partitions are considerably smaller than the whole table, index creation time and cost are significantly lower with Shinobi compared to normal auto-tuning approaches. Shinobi represents partitioning of a table with a tree of subsequent split operations. The root node of the tree represents the whole table and

the leaf nodes represent the actual partitions. Each inner node of the tree describes a split into two partitions defined with predicates that can include all columns of the table. Based on a cost model, Shinobi regularly checks for all parents of leaf nodes if it is beneficial to merge their partitions and for all leaf nodes if it is beneficial to split them.

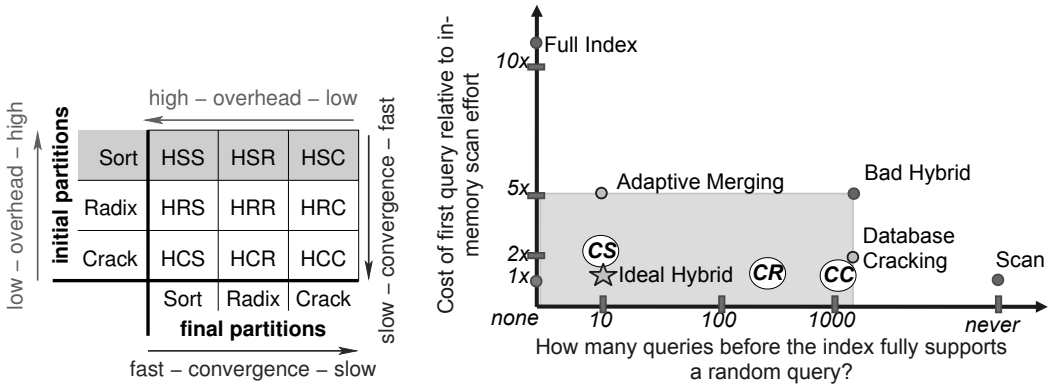
The idea of Shinobi is appealing, because it effectively avoids unnecessary indexing of data in a very simple way. However, implementing partial indexing based on range partitions also has drawbacks. First, the relevance of a tuple regarding a specific index is not necessarily a function of the values of the tuple. For example, the name of a product alone does not drive the interest in the product; it is the advertisement behind it. Second, a tuple is not likely to be equally interesting in each column. In the same example, cheap products may often be queried by price, whereas specifically advertised products may often be queried by their name. With Shinobi, the index tuning of all columns of a table is based on the partitioning of the table. This can make it hard for Shinobi to find an appropriate partitioning, and may result in a very large partitioning tree with very small partitions.

5.1.6 Adaptive Indexing

Adaptive indexing integrates index creation, maintenance, and tuning directly into query processing. Queries themselves build indexes incrementally. An active tuning component becomes obsolete; only a supervisor component is necessary to keep control of indexes' resource consumption. Instead of analyzing the workload for index tuning decisions, adaptive indexing lets the workload actively drive the index tuning.

The first adaptive indexing approach was developed for binary association tables (BATs) – the in-memory storage primitive of the columnar database MonetDB [Boncz and Kersten, 1999, Manegold et al., 2000]. The technique has become widely known as *database cracking* [Kersten and Manegold, 2005, Idreos et al., 2007]. Initially, BATs are unordered, so that point and range queries have to scan the complete BAT to find all matching entries. The idea of database cracking is to make use of what a query finds out to help subsequent queries. Specifically, a query range partitions (cracks) the BAT it scans. The query predicate is used as the partitioning predicate. The BAT is not partitioned physically but partially ordered to reflect the partitioning. All partition definitions are maintained in a so called cracker index, which is typically a tree structure. In memory, cracking poses only a small overhead on the query but immediately helps subsequent queries, which can exploit the partitioning to reduce scanning cost. Subsequent queries continue cracking the partitions they have to scan, so that the partitioning refines where the BAT is queried most. Depending on the value ranges queried, the cracker index converges to a partial or full index.

An extension called *sideways cracking* helps to keep the partitioning of multiple BATs of a table in sync [Idreos et al., 2009]. Particularly in a column store, this is necessary to reduce the tuple reconstruction cost if queries project to additional columns and if queries have predicates on multiple columns of a table. Similarly to



(a) Refinement operations.

(b) Search space of techniques.

Figure 5.3: Hybrids of database cracking and adaptive merging. *Source:* [Idreos et al., 2011]

Shinobi, sideways cracking makes the index tuning of a single column depending on the other columns of the table, with the same drawbacks.

A more recent work addresses the decision making of database cracking. It aims at achieving an index that is more robust towards workload changes. Instead of a strictly query-driven cracking, the so called *stochastic cracking* [Halim et al., 2012] introduces partially arbitrary decisions into the cracking process. Generally, cracking is designed for in-memory column stores and its concepts cannot be readily transferred to block-oriented disk-based database systems.

Inspired by database cracking, Graefe and Kuno developed *adaptive merging* as an adaptive indexing technique for disked-based database systems [Graefe and Kuno, 2010a,b]. Adaptive merging builds on B-trees [Bayer and McCreight, 1972], the most common kind of index structure in disk-based database systems. The concept can be applied to primary as well as secondary indexes. The starting point of adaptive merging is a partitioned B-tree [Graefe, 2003]. Each partition is sorted. In the beginning, the tree will typically have many partitions, all with overlapping key ranges. For instance, the system could create one index partition per disk page. Such an initial index creation is considerably cheaper than the creation of a complete unpartitioned B-tree. Queries that use a partitioned B-tree can efficiently find the matching values within a partition, but have to scan all partitions to find all matches. With adaptive merging, queries incrementally refine the partitioned B-tree by merging the matching index entries into a clean partition. This can be done very efficiently by scanning the B-tree partitions in an interleaved way. Query after query, the clean partition grows into a complete index. If the workload addresses only a subset of values, the clean partition will resemble a partial index. However, the other partitions remain and queries still have to check them. Storage-wise a partitioned B-tree also eats up as much space as a complete index.

Database cracking and adaptive merging share some commonalities. Most obviously, both techniques work on partitions. They create initial partitions and then refine them with every query towards a complete index. Both concepts differ, however, in the operations they use for refinement. Database cracking relies on cracking of partitions while adaptive merging builds on sorting. Radix clustering is a third operation that could be used for refinement. These observations suggest a set of hybrid approaches as presented in [Idreos et al., 2011]. All hybrid techniques have the same base procedure. Developed for the main-memory column store database MonetDB, they started splitting a column into initial unsorted partitions. Every query refines the existing partitions, extracts the matching entries and merges them into a new final partition. The hybrids differ in which refinement operations they apply to initial partitions and final partitions: cracking, radix clustering, or sorting. Depending on the choice, the hybrids exhibit different performance in initial overhead and convergence speed towards a complete index, as shown in Figure 5.3(a). With a combination of cracking and sorting (HCS/CS) the authors claim to get very close to an ideal hybrid regarding initial overhead and convergence; illustrated in Figure 5.3(b).

Database cracking, adaptive merging, and their hybrids are very apealing techniques. They embed incremental index creation directly into the query processing, allowing a database system to create indexes when and where needed along the way, with reasonable overhead. For a complete self-managed index infrastructure, however, these approaches miss two essential things. First, there is no index selection; an index is simply created on any column which is queried. Second, the techniques do not provide any possibility to incrementally discard index information, once it is not needed anymore. In the long run this will result in a large number of complete indexes. In the main-memory column store setting, where all indexes are primary indexes, the number of indexes is limited to the number of columns and only the directory part of the index requires additional storage space. For secondary indexes, though, the techniques are not feasible. Every secondary index would start with a complete copy of the column it indexes, effectively doubling the required storage resources. An unreasonable large number of secondary indexes also increases the index maintenance cost significantly.

5.1.7 Summary

Of all index tuning techniques, adaptive indexing techniques, i.e. database cracking, adaptive merging, and their hybrids as well as the partial indexing tool Shinobi are the most promising approaches for a self-managed indexing infrastructure. Nevertheless, they do not provide a satisfying solution. Database cracking and adaptive merging have two considerable drawbacks. First, they lack a supervisor component, which sets priorities on where to invest the limited system resources on indexing. Second, they are a cul-de-sac with complete indexes at its end. Shinobi is a more comprehensive approach in this respect. It allows de-indexing tuples and is suitable for secondary indexes. However, being based on partitioning of tables, Shinobi makes the index

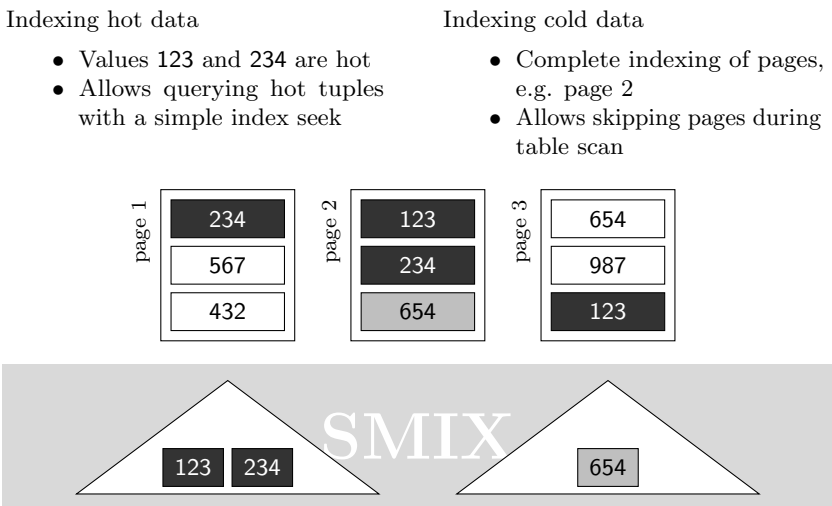


Figure 5.4: Indexing hot and cold data.

tuning on a single column depending on the other columns of the table. The self-managed index infrastructure that we propose avoids these drawbacks.

5.2 SMIX Overview

We propose a novel, adaptable, fine-grained, autonomous indexing infrastructure for secondary indexes in disk-based row stores. It is based on the Self-Managing Index (SMIX), a new access path that partially indexes the column it is working on. Like a table scan, a SMIX is available on every column by default. Like an index, it maintains access information in a secondary data structure for faster predicate evaluation. To remain lean, a SMIX constantly adapts the set of indexed tuples to the workload.

The partial index information helps to speed up queries in two ways:

Indexing hot data If all tuples that fulfill the queried predicate are indexed, the tuples answering the query can be directly discovered with the index. This is the desired case, because a table scan can be avoided. The more queries fall into this category, the better the partial index is adapted to the workload.

Indexing cold data If all tuples in a page are indexed in a partial index, a table scan can skip this fully indexed page. To make sure all tuples matching the queried predicate are found, the database system has to combine the table scan's result set with the result set of the partial index. Hence, skipping a lot of pages during a table scan helps to lower query execution cost in situations where the partial index is not well adapted to the workload and many table scans are necessary to answer queries.

Figure 5.4 illustrates the two cases. The SMIX access path covers both cases by completely indexing (1) the most queried values, to answer most of the common

queries efficiently, and (2) a proportion of pages, to speed up table scans; both in separate index structures. This way, a SMIX is able to adapt to a workload while quickly leveraging collected index information.

As the SMIX is a default access path, it automatically collects index information on potentially every column. Two principles keep the SMIX population of a database from exceeding a configurable global resource limit. (1) Every SMIX has an individual resource quota and it is able to displace less queried index entries to lower its resource usage. (2) All SMIXs compete for the globally granted resources, so that infrequently accessed index information automatically drops out of the system.

The SMIX indexing infrastructure comes with only a very few configuration knobs, mainly the amount of resources that can be used for indexing. SMIXs distribute the heavy lifting of index creation over time and focus index creation on the data of interest. Furthermore, the approach does not involve expensive what-if calls to the query optimizer. This way, SMIXs reduce the required user interaction dramatically without sacrificing performance by missing indexing opportunities or imposing too much overhead on the DBS.

The remainder of this chapter is structured as follows. We continue this overview with an introductory example to illustrate our approach (Section 5.2.1) and an outline of the general SMIX architecture (Section 5.2.2). We detail the SMIX concepts in the following two sections. Section 5.3 details the SMIX access path; how it collects, maintains, and displaces index information. Section 5.4 discusses the global management of all SMIXs present in a system. An extension to SMIX that improves the management of cold data indexing is presented in Section 5.5. We conducted experiments to evaluate the SMIX approach and present the results in Section 5.6.

5.2.1 Introductory Example

Before outlining the details of the SMIX approach, we illustrate the general idea of the SMIX access path with an example. A SMIX consists of three data structures: two tree-structured indexes – the *Partial Index* and the *Index Buffer* – and a list of counters – the *Counter Table*. We describe all three in detail later.

For the example shown in Figure 5.5, we use a table for smartphone sales of cities in the US. Let this table consist of 13 tuples stored in six pages. The table was not queried before and is now hit by three consecutive queries on the same column for which the optimizer decides to use a SMIX scan. All three queries select a new value, not queried before. Before the first query, the SMIX is uninitialized and the data structures do not exist at all and will be created with the first query. The figure shows the state of the SMIX's data structures after every query.

For the first query on NY, the SMIX scans the complete table. While scanning the table, the SMIX inserts the three qualifying tuples into the Partial Index. During the first table scan, the SMIX neither skips any pages nor indexes pages into the Index Buffer, instead it initializes the Counter Table with the number of tuples remaining unindexed in each page.

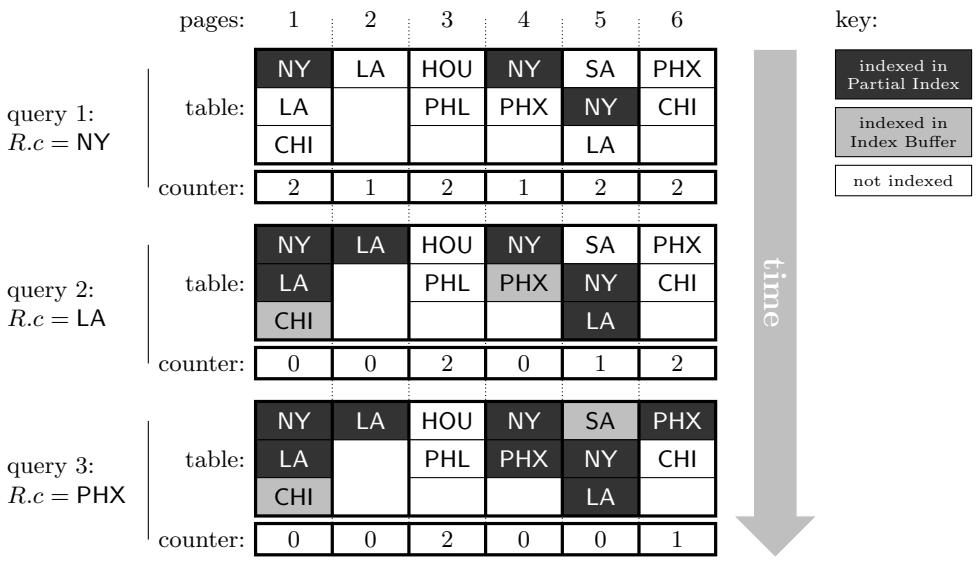


Figure 5.5: SMIX example.

For the second query on LA, three tuples qualify and are indexed into the Partial Index. Since none of the pages is fully indexed yet, no page can be skipped during the table scan. However, with the Counter Table now set, the SMIX can decide before the table scan for which pages it wants to complete the indexation; let it decide on pages 1, 2, and 4. In consequence, the SMIX additionally indexes the two non-qualifying tuples in these pages in the Index Buffer. After the second SMIX scan, six tuples are indexed in the Partial Index, two tuples are indexed in the Index Buffer, and the three pages 1, 2, and 4 are completely indexed. The Counter Table has been updated accordingly.

For the third query on PHX, the SMIX again scans the table. This time, the table scan can skip the completely indexed pages 1, 2, and 4. In the remaining pages, the scan finds two qualifying tuples and indexes them in the Partial Index. Additionally, the SMIX decides to complete indexation for page 5, which has only one unindexed tuple left. The SMIX also scans the Index Buffer, to check for qualifying tuples that may be missed during the table scan by skipping pages. Both resulting tuple streams – of the table scan and the Index Buffer scan – are merged together to form the result of the SMIX scan. Since all tuples found by the Index Buffer scan are qualifying tuples for the queried value, the SMIX removes them from the Index Buffer and adds them to the Partial Index. In the example, this is the case for the second tuple in page 4. After the third SMIX scan, eight tuples are indexed in the Partial Index, two tuples are indexed in the Index Buffer, and the four pages 1, 2, 4, and 5 are completely indexed.

As can be seen in the example, skipping of pages does not result in false negatives. Only completely indexed pages are skipped. For completely indexed pages, all tuples are indexed either in the Partial Index or in the Index Buffer. The Partial Index is

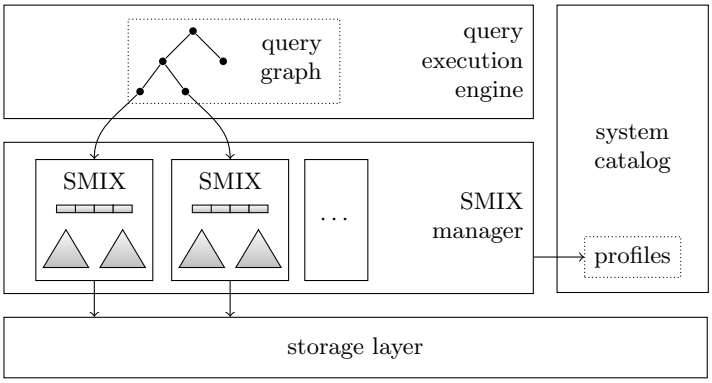


Figure 5.6: SMIX in database architecture.

always scanned; the Index Buffer will be scanned additionally if the Partial Index was negative. Consequently, matching tuples will be discovered no matter where they are indexed.

5.2.2 Architecture

The core idea of SMIX is a new default access path that adapts itself to the workload. This requires two novel components in the database architecture shown in Figure 5.6. The first component is the SMIX itself, the new self-managing access path. The second component is the SMIX manager, which supervises the SMIX population in the system.

A SMIX combines the abilities of a traditional table scan and a traditional index scan in a single access path. Like a traditional table scan, a SMIX acts as a default built-in access path, which is available on every column and does not have to be created explicitly. Like a traditional index, a SMIX incorporates index information, which allows reducing page accesses for queries significantly. Implementation-wise, a SMIX even reuses the logic of these traditional access paths. A SMIX autonomously collects index information based on the tuples that are accessed by the workload. It directly leverages this collected information for the next accesses, even if these accesses relate to other tuples. Additionally, a SMIX not only collects new index information, a SMIX also discards index information that turns out to be less useful. Therefore, a SMIX adapts to the data workload and is also able to control its use of storage and memory resources.

SMIXs co-exist with traditional access paths in the system. The query optimizer still decides which access path to take for a specific query. It applies two general rules for the access path selection: (1) It always chooses a SMIX scan over a table scan, if the optimizer would take a traditional index on this column, because a SMIX can quickly adapt to better performance. (2) It always chooses a traditional index scan over the SMIX scan if a traditional index is present to avoid creating redundant index information. If multicolumn indexes are present on the queried column, the optimizer

relies on traditional statistics-based decision rules. In order to accomplish that, every SMIX maintains statistics about itself in the system catalog, similar to traditional index and table statistics.

SMIXs are query-driven; they are not created explicitly. A SMIX that was never accessed does not consume any space. Each indexable column has a catalog entry indicating if it has an initialized SMIX present. As the SMIX scan is a default access-path a query can utilize a SMIX scan on a column even if the column's SMIX has not been initialized. The first SMIX scan on a column will initialize the SMIX on that column.

The SMIX manager is the supervisor component for all SMIXs in the system. Since SMIXs are automatically created and allocate new storage and memory resource on their own, they need to be controlled to not exceed globally available resources. The SMIX manager collects access statistics for every SMIX. Based on these statistics, it defines resource quotas for SMIXs and enforces them.

The globally available resources for indexing are storage spaces, where the index information is stored. For SMIXs, we distinguish two types of storage spaces: (1) the *index space*, (2) the *buffer space*. The index space represents disk resources; it offers persistency and supports crash recovery. A SMIX stores its established indexing information here, which has proven valuable for the workload. In contrast, the buffer space represents main memory resources; it is transient and does not support crash recovery. A SMIX stores supporting structures that contain less valuable indexing information in the buffer space. Especially when a SMIX is barely adapted to the workload, it makes heavy use of these supporting structures. Hence, the main-memory-based buffer space allows a faster adaptation at lower cost compared to disk. The SMIX manager assigns quotas for index space and buffer space to each SMIX, while the absolute size of index space and buffer space is configured by the DBA.

Next, in Section 5.3, we dive into detail on the SMIX design, especially which data structures it uses and how it operates. Section 5.4 provides a detailed description of the SMIX manager, on how quotas are determined and enforced. Additionally, we discuss an extension of the SMIX concept exploiting partitioned Index Buffers in Section 5.5.

5.3 SMIX Access Path

In this section we describe the functioning of the adaptive indexation logic of a single SMIX, which works on a single column of a relation. First we introduce the data structures that are utilized for the indexation process. This is followed by the state model, which specifies the different modes of operation a SMIX can be in. Then we show how the data structures and the state model work together. Further, we discuss how to reverse the indexation process, displace index information and free resources, which then can be claimed by other SMIXs. Finally, we outline how SMIX are maintained during DML operations.

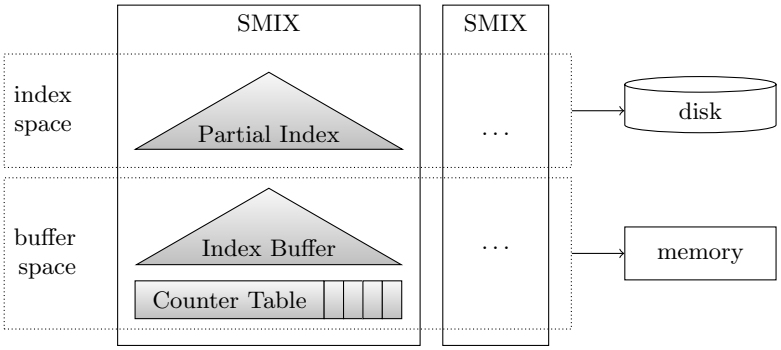


Figure 5.7: Data Structures of a SMIX.

5.3.1 Data Structures

A SMIX consists of three data structures: the *Partial Index*, the *Index Buffer*, and the *Counter Table*. Depending on their usage, each of these structures is either stored inside the index space or the buffer space, as illustrated in Figure 5.7. The following describes the three data structures and their usage in more detail.

Partial Index The Partial Index is a conventional B*-tree [Comer, 1979], which resides in the index space. The Partial Index completely indexes the most queried values. For values with qualifying tuples, the Partial Index holds references on all these qualifying tuples. For values without any qualifying tuples, the Partial Index holds a single null reference. Either way, the SMIX can serve queries on values present in Partial Index by a single Partial Index scan. The SMIX inserts values into the Partial Index when they are queried. To control its size, the SMIX also removes infrequently queried values from the Partial Index. By indexing only the most frequently queried values, the Partial Index reflects the current workload of the database – similar to a cache or a buffer. Thus, once adapted to the workload, the SMIX serves the majority of its queries by a single efficient Partial Index scan.

Index Buffer The Index Buffer is also a conventional B*-tree and is stored inside the transient buffer space. The Index Buffer completely indexes the remaining tuples of a page that are not already indexed by the Partial Index. This way, the Index Buffer complements the Partial Index; all tuples of pages referenced in the Index Buffer are indexed either in the Partial Index or in the Index Buffer. To avoid redundant indexing information, the tuple reference set of the Partial Index and the Index Buffer are disjoint. With the help of the Index Buffer, the SMIX can increase the number of completely indexed pages significantly. The SMIX can safely skip these completely indexed pages during a table scan without risking false negatives. All potential result tuples missed in the table scan can be discovered by a Partial Index or an Index Buffer scan. While still adapting to the workload, the SMIX uses the Index Buffer to speed up the table scans still required for a large share of queries. Thus, the Index

Buffer serves as a temporary supporting structure in times of workload changes. The Index Buffer can be seen also as a specialized buffer pool, which buffers data in a processing-oriented form.

Counter Table The Counter Table is a list of counters located inside the buffer space. The list contains a counter for each page of the table the SMIX serves. Each counter indicates how many tuples in its corresponding page are neither indexed by the Partial Index nor by the Index Buffer. The Counter Table serves two purposes: (1) it helps to quickly identify pages that are most worthwhile to be indexed in the Index Buffer (pages with a low counter greater than zero); (2) it allows to easily identify pages that can be skipped by a table scan (pages with a counter equal to zero). The SMIX initializes the Counter Table with the first table scan it has to perform. Subsequently, the SMIX maintains the Counter Table incrementally. The memory the Counter Table consumes depends on the accuracy with which pages are monitored a, the maximum number of tuples that fit in a page n, and the total number of pages p, so that the size of the Counter Table equals $\frac{N\log_2(n)}{a}$. Note that the accuracy $a = \frac{1}{x}$ with x being a positive integer.

Please note that conventional B*-trees are not a necessity for the SMIX concept. A Partial Index or an Index Buffer built as a hash index, a spatial index, or the use of cache-optimized index structures such as the CSB+-Tree [Rao and Ross, 2000] is equally suitable.

5.3.2 State Model

The operational mode of a SMIX depends on its need to adapt itself to the workload. A SMIX is well adapted to the workload if it can serve the majority of queries with an efficient Partial Index scan, i.e. if its behavior resembles that of a standard index. Accordingly, we define the percentage of the recent queries that could be answered with a Partial Index scan as the Partial Index hit rate. If the Partial Index hit rate of a SMIX is low, the SMIX will operate in *unstable* mode and try to adapt to the workload. If the Partial Index hit rate is above a given threshold θ, the SMIX operates in *stable* mode. Depending on its Partial Index hit rate, a SMIX switches between both states. A SMIX measures its Partial Index hit rate h using a ring bitmap B with a configurable time frame t so that $h = |\{x|x \in B \wedge x = 1\}| \cdot t^{-1}$. Table 5.2 summarizes the most important characteristics of the two operational states.

Unstable State In the unstable state, the SMIX has a low Partial Index hit rate, which results directly from the low number of SMIX accesses that are processed by a Partial Index scan. This means that the index needs to adapt to the current workload. The SMIX builds up and maintains the Partial Index and the additional Index Buffer to temporarily speed-up the high number of table scans. In effect, the SMIX occupies resources in the index space as well as in the buffer space. To force the SMIX into the stable state in a reasonable amount of time, the SMIX desists displacing entries

Table 5.2: State characteristics of a SMIX.

	Unstable	Stable
Partial Index hit rate	low	high
% index scans	low	high
% table scans	high	low
Need to adapt	high	low
Index Buffer present	yes	no
Counter Table present	yes	no
Partial Index build-up	yes	yes
Automatic displacement on Partial Index	no	yes

from the Partial Index automatically. (Nevertheless, the SMIX displaces Partial Index entries if it is forced to by the SMIX manager.)

Stable State The stable state is characterized by a high Partial Index hit rate, which means that the SMIX serves most queries with an efficient Partial Index scan. This implies a low probability of expensive table scans. Therefore, the SMIX discards the Index Buffer and the Counter Table and solely relies on the Partial Index. In effect, the SMIX occupies resources in the index space only. The SMIX further builds up the Partial Index, in case unindexed values are queried. Additionally, it automatically displaces the most infrequently accessed Partial Index entries. Since these entries obviously do not fit the current workload anymore, they are not worth keeping. The goal of a SMIX is to get into the stable state, to serve the current workload most efficiently.

5.3.3 SMIX Scan

Traditionally, the database system accesses the requested data either via a full table scan or by scanning a full-column index. Our approach combines both access paths in a single SMIX scan. During the SMIX scan, the Partial Index and the Index Buffer gather indexing information, which is used to speed up subsequent SMIX scans.

A SMIX scan consists of two phases. In the first phase, the SMIX scans the Partial Index for the queried value. If the value is found in the Partial Index, the result of the Partial Index scan answers the query. In our implementation, the Partial Index scan is a traditional index scan over a B*-Tree. In the second phase, the SMIX scans the Index Buffer and the table for the queried value in case the Partial Index scan was negative. Like the Partial Index scan, the Index Buffer scan is a traditional index scan over a B*-Tree in our case. The table scan, though, performs three additional actions besides searching for tuples with the queried value. (1) The table scan skips all pages with a Counter Table entry equal to zero. (2) The table scan indexes all unindexed tuples of pages with the lowest Counter Table entries in the Index Buffer. (3) The table scan indexes all qualifying tuples in the Partial Index.

Algorithm 5 SMIX scan.

```
1:  procedure SmixScan(R, q, IX, B, C)                    ▷ R: set of pages to scan
2:                                                        ▷ q: queried predicate
3:                                                        ▷ IX: Partial Index
4:                                                        ▷ B: Index Buffer
5:                                                        ▷ C: Counter Table
6:      Q ← IX.Scan(q)                                    ▷ Partial Index scan
7:      if Q = ∅ then                                     ▷ If no matches found in Partial Index
8:          I ← SelectPagesForIndexBuffer(C)              ▷ Select pages for Index Buffer
9:          for t ∈ B do                                  ▷ Index Buffer scan
10:             if q(t) then                              ▷ If tuple matches predicate
11:                 Q ← Q ∪ {t}                           ▷ Add tuple to result set
12:                 B.Remove(t)                           ▷ Remove tuple to Index Buffer
13:                 IX.Add(t)                             ▷ Add tuple to Partial Index
14:         for p ∈ R with C[p] > 0 do                    ▷ Table scan
15:             for t ∈ p do                              ▷ Page scan
16:                 if q(t) then                          ▷ If tuple matches predicate
17:                     Q ← Q ∪ {t}                       ▷ Add tuple to result set
18:                     IX.Add(t)                         ▷ Add tuple to Partial Index
19:                     C[p] --                           ▷ Decrement Counter Table
20:                 else if p ∈ I ∧ t ∉ IX then           ▷ If page should be indexed
21:                     B.Add(t)                          ▷ Add tuple to Index Buffer
22:                     C[p] --                           ▷ Decrement Counter Table
23:     return Q                                          ▷ Return result set
```

Algorithm 5 shows the SMIX scan in detail. Given is a query with predicate q, operating on the pages in R, the Partial Index IX, the Index Buffer B, and the Counter Table C. The algorithm has two phases. In the first phase, the SMIX scans its Partial Index (line 6). If the Partial Index contains entries matching q, the SMIX scan is done and returns the result of the Partial Index scan. If the Partial Index result is empty instead, the SMIX performs the second phase. As preparation, the SMIX determines the pages that should be fully indexed during this scan (line 8). Thereafter, the SMIX scans its Index Buffer (line 9) and the table (line 14). Index Buffer and table can be scanned in parallel. However, this may cause touching tuples twice, because the table scan adds entries to the Index Buffer. The Index Buffer scan adds all qualifying tuples to the result and moves them to the Partial Index. The table scan also adds all qualifying tuples to the result and to the Partial Index. Additionally, if the scanned page was selected to be completely indexed, the table scan adds all non-qualifying tuples that are not already indexed in the Partial Index to the Index Buffer. For all tuples added to either of the indexes, the scan decrements the page's counter in the Counter Table. Finally, the SMIX returns the result.

A SMIX scan operates in the unstable state as described. In the stable state, the SMIX performs only a table scan in case the Partial Index scan was negative. It does not scan or maintain an Index Buffer. For range predicates the SMIX scan always has to perform a table scan, because the Partial Index may not cover the complete range.

The number of pages the SMIX indexes in the Index Buffer in each table scan depends on the Partial Index hit rate. The lower the Partial Index hit rate is, the more table scans require speedup, the more pages we want to be completely indexed after the next table scan. Accordingly, the SMIX determines the number of pages as $\frac{\theta-h}{\theta} \cdot \frac{b}{x}$, where h is the SMIX's current Partial Index hit rate, θ is the configured threshold for the stable state, b is the total number of pages in the table, and x is a damping factor. For $x = 1$ the SMIX would complete the indexing of all pages in the first scan after initialization, for $x = 2$ it is half of the pages, for $x = 3$ one third and so on. We found $x = 2$ to be a practical setting. Note that this approach quickly increases the number of pages that can be skipped in the table scan; it has no influence on the improvement of the Partial Index hit rate, nor does it facilitate reaching the stable state.

5.3.4 Displacement

While SMIXs grow incrementally, SMIXs are also able to shrink. Shrinking is crucial for the constant adaptation to a shifting workload. By displacing index entries that are not worth keeping for the current workload or an evolved schema, SMIX frees storage and maintenance resources, which can be spent on index entries that are more valuable for the current workload. Displacement is triggered in two ways: *forced displacement* and *automatic displacement*. The SMIX manager orders forced displacement to keep a SMIX within its resource quotas. Every time a SMIX indexes new data and requires more space, the SMIX manager checks the quotas and triggers a forced displacement if required. The SMIX itself performs automatic displacement to remove scarcely used index information. With every query a SMIX processes in the stable state, the SMIX checks if some index information can be displaced. A SMIX implements different displacement strategies for its Partial Index and its Index Buffer. Both strategies are explained in the following.

Partial Index Displacement The Partial Index contains the index entries most valuable to the current workload. Once the SMIX is in the stable state, it reflects the current workload and serves the majority of queries. However, when the workload shifts, Partial Index entries created in the previous workload episode may not be valuable to the current workload episode. To discard these entries from the Partial Index in an efficient way, the SMIX can remove the least frequently accessed leaf nodes from the Partial Index. Discarding a leaf node removes the range of values from the Partial Index that have their entries in that node. This may also include valuable entries, but the overhead of tracking the value of single entries would be prohibitively high and truly valuable entries will return soon.

Table 5.3: SMIX maintenance.

		$t \in IX$		$t \notin IX$	
		$t' \in IX$	$t' \notin IX$	$t' \in IX$	$t' \notin IX$
		IX.Update(t, t')	IX.Remove(t)	IX.Add(t')	-
$p \in B$	$p' \in B$	-	B.Add(t')	B.Remove(t)	B.Update(t, t')
	$p' \notin B$	-	$C[p']$++	B.Remove(t)	B.Remove(t), $C[p']$++
$p \notin B$	$p' \in B$	-	B.Add(t')	$C[p]$--	B.Add(t'), $C[p]$--
	$p' \notin B$	-	$C[p']$++	$C[p]$--	$C[p]$--, $C[p']$++

More specifically, the Partial Index discards a leaf node in five steps. (1) It selects the leaf node that should be displaced. (2) It removes the referencing entry from the leaf node's parent node. (3) It frees the page the leaf node was stored in. (4) It removes overlapping entries from the neighboring leaf nodes. (5) It increments Counter Table entries accordingly for every page that is referenced in the leaf node and the overlapping entries.

The crucial step is the selection of the leaf node to discard. To discard the leaf node that is of least worth to the current workload, the SMIX selects the node least recently used by a query. For that purpose the SMIX maintains the *historic mean access interval* (Δ) and the *current mean access interval* ($\hat{\Delta}$) for all leaf nodes of its Partial Index. The higher both measures, the less a node was recently used. Both measures are explained in detail in Section 5.4.1. For forced displacement, the SMIX removes the leaf nodes with the highest $\hat{\Delta}$ values. For automatic displacement, the SMIX removes leaf nodes whose $\hat{\Delta}$ exceed its Δ by a factor α: $\hat{\Delta}/\Delta > \alpha$. The factor α controls the aggressiveness of automatic displacement on Partial Indexes; the lower α, the more aggressively the SMIX discards leaf nodes.

Index Buffer Displacement The Index Buffer is a supporting structure held in memory only, which helps a SMIX during time of adaptation. More specifically, its index entries are gap fillers to complement the entries of the Partial Index so that the pages are fully indexed and can be skipped in a table scan. Discarding a whole node of Index Buffer entries would cause many pages not to be completely indexed anymore, which would defeat the purpose of the Index Buffer without freeing many resources. Thus such a fine-grained displacement is not possible for the Index Buffer. In consequence, the SMIX simply discards the whole Index Buffer. Discarding the Index Buffer does not hurt the stability of a SMIX, it merely slows the SMIX on the next table scans, because it can skip less pages. A SMIX discards its Index Buffer if it is ordered to do so by the SMIX manager (forced displacement), or if it enters the stable state (automatic displacement), since a SMIX does not maintain an Index Buffer in the stable state (cf. Section 5.3.2).

5.3.5 Maintenance

A SMIX maintains the Partial Index, Index Buffer, and Counter Table during inserts, updates, and deletes. Which operation the SMIX has to perform depends on (1) if the old tuple t was in the Partial Index, (2) if the updated tuple t' will be in the Partial Index, (3) if the old page p that contained the updated tuple is covered by the Index Buffer, and (4) if the new page p' that will contain the new tuple is covered by the Index Buffer. Table 5.3 lists the different maintenance scenarios with necessary operations.

5.4 SMIX Manager

With the SMIXs acting individually and independent from each other, the SMIX manager takes on the task of supervising the whole population of SMIXs present in a system. The main goal of the SMIX manager is to ensure that the globally granted resources for SMIXs are not exceeded. The SMIX manager determines resource quotas for every SMIX and enforces these quotas.

5.4.1 Resource Quotas

Resource quotas define how much storage and memory each SMIX can occupy in the index space and the buffer space, respectively. Depending on its state, a SMIX gets a specific share of both storage and memory (unstable), or only of storage (stable). For each SMIX, the SMIX manager continuously calculates the shares. The shares of a SMIX reflect (1) how often the SMIX is used and (2) the size of the column the SMIX indexes.

We determine the usage of a SMIX with the same measures used for the displacement in a Partial Index: the *historic mean access interval* Δ and the *current mean access interval* $\hat{\Delta}$. For both, consider Figure 5.8. Looking at a history of all accesses to all SMIXs in a system, we can determine how frequently a particular SMIX is used by averaging the length of the intervals between accesses to this SMIX. For this, the SMIX manager maintains a history $H = [\delta_1, \ldots, \delta_l]$ of the recent access interval lengths δ_i for each SMIX, so that δ_1 presents the most recent access interval and l is the maximum history length maintained. Additionally, every SMIX has a counter δ_{now}, which the SMIX manager increases with every access to any of the other SMIXs. If a SMIX is accessed, its δ_{now} becomes the new first entry of the history, so that $H = [\delta_{now}, \delta_1, \ldots, \delta_{l-1}]$. The historic mean access interval Δ of a SMIX is the mean of its history, $\Delta = (\delta_1 + \ldots + \delta_l) \cdot l^{-1}$. The smaller Δ, the more frequently a SMIX is accessed. In the example shown in Figure 5.8, the considered SMIX has a history of $[7, 3, 8, 6]$, $\delta_{now} = 4$ and $\Delta = 6$.

The historic mean access interval Δ already gives a good measure of how frequently a SMIX is used. However, Δ only reflects the past, i.e. accesses that happened. If a SMIX is not accessed anymore because of a workload change, it will keep a small Δ. To consider also a SMIX's current interval between the last access and the next

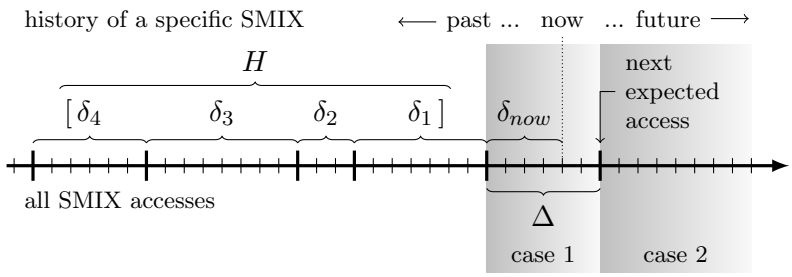


Figure 5.8: Mean access interval measures.

access to come, we distinguish two cases: either (1) the expected point for the next access is still to come ($\delta_{now} \leq \Delta$), or (2) has already past ($\delta_{now} > \Delta$). In the first case, we stick with the expected interval and assume the current interval to equal Δ. In the second case, we know the expectation is wrong and the current interval is at least as long as δ_{now}. The current mean access interval $\hat{\Delta}$ of a SMIX is the mean of its history including the current interval:

$$\hat{\Delta} = \begin{cases} \Delta & \delta_{now} \leq \Delta \\ \frac{\delta_{now}+\delta_1+\ldots+\delta_h}{h+1} & \delta_{now} > \Delta \end{cases}$$

The size s of the SMIX indexed column is calculated from the number of tuples k in the corresponding table and the column width l: $s = kl$. A SMIX is worth higher shares than other SMIXs if it is accessed more frequently and if it has to cover a larger amount of data compared to other SMIXs. Hence, in a SMIX population P, the share of a SMIX is:

$$\omega = \frac{\hat{\Delta}^{-1} s}{\sum_P \hat{\Delta}^{-1} s}$$

The population P encompasses all SMIXs for the index space and all unstable SMIXs for the buffer space, resulting in two shares ω_{IS} and ω_{BS}, respectively. Based on these two shares, the SMIX manager grants each SMIX a number of pages in the index space and in the buffer space as resource quotas.

5.4.2 Quota Enforcement

With the help of the quotas, the SMIX manager determines which SMIX can grow and which SMIX has to shrink. However, because of the different displacement characteristics, the enforcement of the individual quotas for index space and buffer space differs too. In the following, we describe the enforcement for both spaces in detail.

Quota Enforcement in Index Space The index space hosts Partial Indexes only. Partial Indexes grow and shrink moderately, which allows an optimistic enforcement of the quotas. Requests for new Partial Index pages are always granted to a SMIX

regardless of its quota. If the available index space is exhausted, the SMIX manager will order forced single-page displacements until the required number of pages is free. For every displacement, the SMIX manager selects a SMIX following two rules: (1) If SMIXs exceed their quota, the SMIX manager will pick the SMIX with the largest relative excess. (2) If multiple SMIXs have the same relative excess, the SMIX manager will pick the least recently used SMIX among them (highest $\hat{\Delta}$). The selected SMIX decides which specific page it will displace (see Section 5.3.4).

Quota Enforcement in Buffer Space The buffer space hosts unstable SMIXs, specifically Counter Tables and Index Buffers. Both account for the quota of a SMIX. Counter Tables change their size only in case of inserts to the table, and are only displaced if the SMIX reaches the stable state. In case the quota of a SMIX is too low to fit its Counter Table, the SMIX manager removes Index Buffer and Counter Table completely. However, the manager keeps determining the SMIX's quota, so that the SMIX may be allowed to initialize again later. In contrast to Counter Tables, Index Buffers change their size rapidly. On the one hand, an Index Buffer grows rapidly if the SMIX is below its quota in the buffer space and completes indexing for many pages in each table scan. On the other hand, an Index Buffer shrinks suddenly to zero size if the SMIX manager forces its displacement. In consequence, the SMIX manager enforces the buffer space quotas pessimistically to avoid excessive displacement. Requests for new Index Buffer pages are granted to a SMIX as long as the SMIX does not exceed its quota. This strategy prevents a SMIX from actively exceeding its quota. However, a SMIX can passively exceed its quota should the quota change. After each recalculation of the buffer space quotas, the SMIX manager determines which SMIXs exceed their quota and orders a forced displacement of one of them. To select the SMIX that will have to displace its Index Buffer, the SMIX manager applies the same two rules as in the index space (largest relative excess and least recently used). A more elaborate management of Index Buffers as an extension to SMIX is discussed in the following section.

5.5 Partitioned Index Buffer

With various SMIXs in a database, multiple Index Buffers are created over time. All Index Buffers reside in the buffer space, which is a share of the database buffer of limited size. The SMIX manager assigns to each SMIX a quota for its Index Buffer. If Index Buffers exceed their quotas, the SMIX manager removes one of them completely from the buffer space with a forced displacement. Obviously, this is not an ideal strategy as many still useful Index Buffer entries are discarded. Partitioned Index Buffers allow the SMIX manager (1) to enforce the quotas with less radical means, and (2) always to have the most benefical entries in an Index Buffer. The general procedure of quota enforcement remains the same, though. The SMIX manager controls the total size of the buffer space before SMIXs add new entries with a tables scan. In case the new entries would cause the buffer space to exceed its allowed size,

the SMIX manager discards index information from the buffer space to maintain the limit.

5.5.1 Precise and Efficient Index Buffer Displacement

The purpose of the Index Buffer is to allow page skipping during a table scan. A single entry in the Index Buffer can reference multiple pages. Further, a single page can be referenced by multiple index entries. Hence, discarding a single entry from the Index Buffer has a double negative effect. First, one or more pages obtain an unindexed tuple and cannot be skipped anymore. Second, all other entries in the Index Buffer referencing these pages occupy memory in the buffer space without creating any benefit. In consequence, discarding single entries from the Index Buffer contradicts the buffer's purpose. Discarding the complete Index Buffer, in contrast, is overacting; an excessive management intervention that goes beyond that what is actually necessary.

For the precise and efficient discarding of entries from an Index Buffer, we partition the B*-Tree of an Index Buffer. Each partition covers P pages of the table, so that, the partitions are disjoint in the sets of pages they reference. For instance, if an index entry in partition 3 references page 8, all other Index Buffer entries that reference page 8 are in partition 3, too. In case the SMIX manager decides to discard index information, it always drops complete partitions from the buffer space. This efficiently discards all entries that reference a set of pages.

Figure 5.9 shows an example with two Index Buffers in the buffer space. The first Index Buffer belongs to the SMIX on column X and is partitioned in three partitions. Each partition indexes $P = 2$ pages. For instance, partition 1 indexes the three tuples in page 1 and page 7 that are not covered by the Partial Index. The second Index Buffer belongs to the SMIX on column A and is partitioned in two partitions. Note that, similar to normal secondary indexes, it is insignificant for the separation of Index Buffers whether the columns are in the same table or not.

5.5.2 Management of Buffer Space

The buffer space is managed based on the benefit and the size of index information. New index information can discard old index information of the same or larger size if it provides more benefit than the old index information. The benefit of index information in the Index Buffer is determined by on two factors: (1) the number of pages covered by the index information, and (2) how frequently the index information is used. The higher these factors are, the more pages the workload can skip. The size of the index information is the amount of memory it requires to be stored. Generally, the number of entries in an index determines the size of the index.

The frequency with which an Index Buffer is used is determined by its current mean access intervals (cf. Section 5.4.1). For the partitioned Index Buffer, the SMIX manager maintains current mean access intervals for Partial Index and Index Buffer with separate access histories. $\hat{\Delta}_{IX}$ is the current mean access interval for a Partial

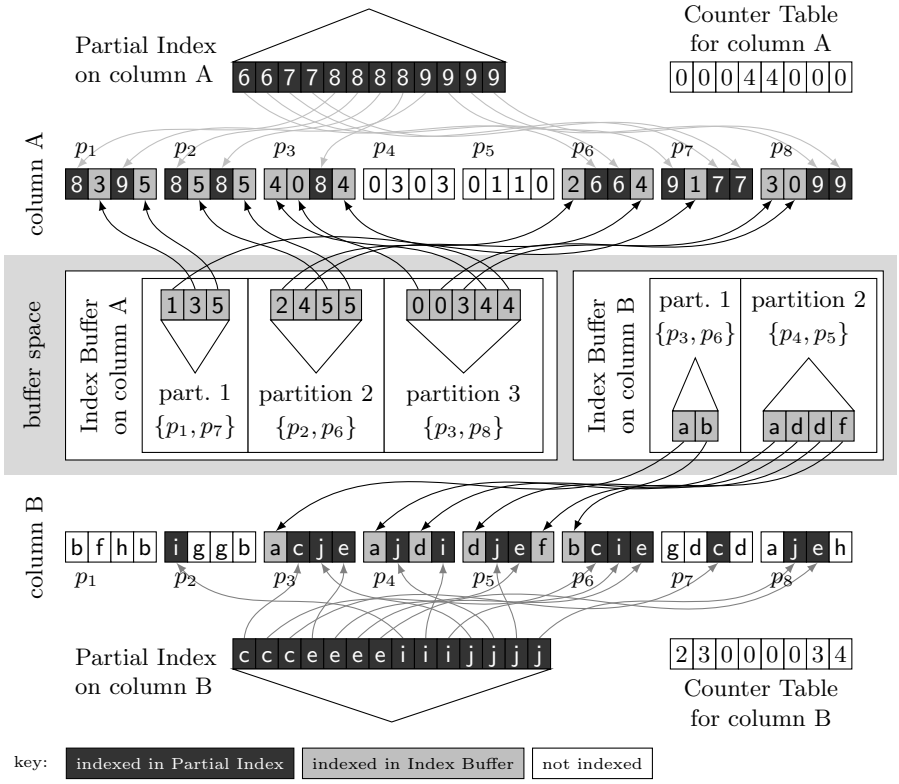


Figure 5.9: Partitioned Index Buffers.

Index, used for the quota determination in the index space. It reflects every access to a SMIX, regardless whether a query hit the Partial Index or not. In contrast, the current mean access interval for an Index Buffer $\hat{\Delta}_B$ reflects only access to the Index Buffer of a SMIX. $\hat{\Delta}_B$ does not take into account queries that a SMIX can answer with the Partial Index because they do not utilize the Index Buffer. To emphasize the difference beween the two, Table 5.4 summarizes the operations performed on the access histories of the SMIXs according to whether the Partial Index of the queried column is hit or not. For simplicity, $H[0]$ takes the role of δ_{now} here. As a consequence of the separate histories, a heavily used, stable SMIX with a well adapted Partial Index will have a low $\hat{\Delta}_{IX}$ but a high $\hat{\Delta}_B$. While for an unstable SMIX $\hat{\Delta}_{IX}$ and $\hat{\Delta}_B$ will be alike. Note, $\hat{\Delta}_{IX}$ is never larger than $\hat{\Delta}_B$.

Further, let the number of pages covered by a partition p be $\mathcal{X}_p$. Then, the benefit of partition p results from $b_p = \mathcal{X}_p \cdot \hat{\Delta}_B^{-1}$ where B is the Index Buffer the partition belongs to. Accordingly, the benefit of an Index Buffer B is the sum of the benefits of its partitions:

$$b_B = \sum_{p \in B} b_p \ .$$

Table 5.4: Operations on access histories.

	SIMX of queried column	SMIX of other columns
Partial Index hit	shift(H_{IX}, +1); $H_{IX}[0] = 0$ $H_B[0]$++	$H_{IX}[0]$++ $H_B[0]$++
no Partial Index hit	shift(H_{IX}, +1); $H_{IX}[0] = 0$ shift(H_B, +1); $H_B[0] = 0$	$H_{IX}[0]$++ $H_B[0]$++

Similar to partitions, the benefit of new index information results from $b_I = |I| \cdot \hat{\Delta}_I^{-1}$, where I is the set of pages to index and $\hat{\Delta}_I$ is the mean access interval of the Index Buffer that will accommodate the new index information.

For the size of index information, we denote the number of entries in a partition p as n_p. Analogously, the size of new index information results from the number of entries to add, denoted as n_I. Based on the counters for unindexed tuples in pages, the system can easily determine n_I as

$$\sum_{s \in I} C[s] \ .$$

Further, let n_F be the free space left in the buffer space.

Before the system adds new index information to the Index Buffer, it has to select the pages it wants to index. At that point the SMIX manager also checks the space bound of the buffer space and triggers displacement of old index information if required. The page selection routine ensures that there is enough buffer space available to index the pages it returns.

The management strategy of the partitioned Index Buffer wants to achieve two conflicting goals. On the one hand the Index Buffer should index as many pages as possible to be useful quickly. On the other hand existing index information should stay in the Index Buffer as long as possible to be present if needed. To balance between both goals, the SMIX manager indexes precisely the right number of pages so that the resulting new index information is more benefical than the old index information that the SMIX manager must discard to clear the space required for the new index information. Additionally, there is a configurable upper bound for new index information per table scan.

Algorithm 6 shows the page selection routine for partitioned Index Buffers. It returns the set I of pages for indexation (line 20) and discards a set D of partitions to ensure that enough space is available (line 19). To determine I and D, the routine iteratively adds partitions to D (line 11). In each iteration the algorithm performs three steps. First, the algorithm determines the available space in the Index Buffer – total size of the partitions in D plus free space (line 8). Second, it selects the set of pages I so that the new index information will fit in the available space (lines 12–17).

Algorithm 6 Select pages for indexation.

```
1:  procedure SelectPagesForIndexBuffer(R, C, S)
2:                                                  ▷ R: set of pages to scan
3:                                                  ▷ C: counter table
4:                                                  ▷ S: set of all partitions in buffer space
5:      D' ← ∅                                      ▷ D': set of collected candidate partitions
6:      I' ← ∅                                      ▷ I': set of collected candidate pages for Index Buffer
7:      repeat                                      ▷ Partition selection loop
8:          n_A ← n_F + Σ_{p∈D'} n_p                ▷ Determine available space
9:          b_{I'} ← |I'| · Δ̂^{-1}_{I'}             ▷ Determine benefit of candidate pages
10:         D ← D'                                  ▷ D: set of partitions to remove
11:         D' ← D' ∪ {SelectNextPartition(S)}      ▷ Add next candidate partition
12:         I ← I'                                  ▷ I: set of pages for Index Buffer
13:         I'' ← I'                                ▷ I'': set of candidate pages for page selection loop
14:         repeat                                  ▷ Page selection loop
15:             I' ← I''
16:             I'' ← I'' ∪ {SelectNextPage(C, R)}  ▷ Add next page candidate
17:         until n_{I''} > n_A ∨ |I''| > I^MAX     ▷ As long as space is available
18:     until b_{I'} ≤ Σ_{p∈D'} b_p ∨ I' = I        ▷ As long as benefit increases
19:     DropPartitions(D)                           ▷ Remove partitions
20:     return I                                    ▷ Return set of pages for Index Buffer
```

Specifically, the algorithm adds pages in ascending order of the counter C as long as

$$n_I \leq n_F + \sum_{p \in D} n_p \ .$$

At most the SMIX indexes I^{MAX} pages during one table scan. Third, the algorithm determines the benefit of the new index information b_I resulting from an indexation of the pages in I (line 9). The algorithm repeats the three steps as long as the benefit of indexing the pages in I is higher than the benefit of the partitions in D

$$b_I > \sum_{p \in D} b_p$$

or until I does not change anymore (line 18).

The SMIX manager selects each partition in D following a two-staged selection algorithm. In the first stage, the algorithm randomly selects an Index Buffer B with the probability

$$\frac{b_B^{-1}}{\sum_{B' \in S \setminus B_N} b_{B'}^{-1}} \ ,$$

where S is the set of all Index Buffers in the buffer space, and B_N is the Index Buffer the new entries should be added to. Index Buffers with a low benefit are more likely

Table 5.5: SMIX base configuration.

Parameter	Value
Size of index space	64 MB
Size of buffer space	128 MB
Threshold θ	95 %
Time frame t	200
Displacement factor α	∞
History length n	3

to be picked. In the second stage, the algorithm selects a partition from that Index Buffer. A possible incomplete partition ($\mathcal{X}_p < P$) has the lowest benefit within an Index Buffer and will be picked first. Afterwards, complete partitions are picked in descending order of their size (n_p), beause they have the same benefit.

5.6 Evaluation

To evaluate the SMIX infrastructure, we conducted a series of experiments. This section presents the results of this evaluation. We start giving an overview of our prototype and describe the setup that we used for our experiments in Section 5.6.1. In Section 5.6.2, we present a basic experiment to illustrate the behavior and the inner workings of a single SMIX, especially its ability to adapt to a changing workload. Based on this experiment, we investigate the influence of the different SMIX parameters in Section 5.6.3. In Section 5.6.4, we evaluate a set of common workload patterns for a single SMIX. To evaluate the SMIX manager, we extend these workload patterns in Section 5.6.5 to a more complex scenario, which involves multiple SMIXs. All these experiments are based on the basic version of SMIX using an unpartitioned Index Buffer. We evaluated the partitioned Index Buffer separately; results are presented in Section 5.6.6.

5.6.1 Setup

We implemented our prototype of SMIX in PostgreSQL 9.0.2. In the prototype, SMIXs are the new default access path available by default on every column. Implementation-wise, our prototype reuses the existing heap scan code and B*-Tree code of PostgreSQL as much as possible. We ran all experiments on an Intel Core i7 processor at 3.4 GHz with 8 GB of DDR3 main memory and a 1 TB hard drive. We used Microsoft Windows 7 64 bit edition as the operating system.

For all experiments, we used a common data setup, which consists of a single table with three INTEGER columns A, B, and C for indexing and one VARCHAR(512) column as payload. The integer columns are the column queried. The payload data represents descriptive attributes of the entities not used in selection predicates. All

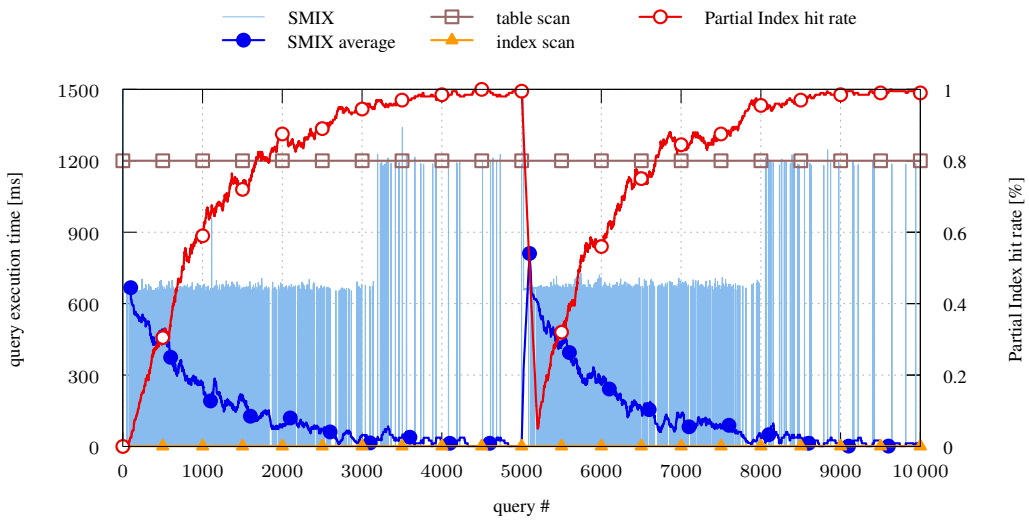


Figure 5.10: Query execution time and Partial Index hit rate.

three integer columns are populated with random values uniformly distributed from 1 to 50 000. The size of the payload values is also uniformly distributed from 1 to 512 byte. We filled the table with 5 000 000 tuples, resulting in an effective table size of 1.5 GB on disk. In the base configuration, the database system is configured to use 256 MB of shared buffers. Furthermore, the base configuration encompass an initial setting for all the parameters listed in Table 5.5. The base configuration applies to all experiments unless stated otherwise.

5.6.2 General Performance

Our initial experiment illustrates the general behavior and performance of a single SMIX and how it adapts to a changing workload. In the experiment we run a workload consisting of two subsequent episodes. Each workload episode is a set of 5000 queries in the form of `SELECT COUNT(*) FROM R WHERE c=`x. For every query, we pick x randomly from an equally distributed continuous range of the domain of column **c**. In the first episode $x \in [1000, 2000]$ and in the second episode $x \in [25\,000, 26\,000]$. We used the base configuration, except the size of the buffer space was reduced to 64 MB. For all 10 000 queries, we measured the execution time, the Partial Index hit rate, the Partial Index size, and the Index Buffer size. We compare our measurements with two baselines: the traditional table scan and the traditional full-column index.

Figure 5.10 shows the execution time and the Partial Index hit rate over the course of the workload. The traditional table scan and the traditional full-column index exhibit a constant execution time over the complete workload of about 1200 ms and 1 ms, respectively. The execution time remains constant, since the two traditional access paths treat every query equally and do not adapt to the workload. In contrast, the execution time of the SMIX varies over the course of the workload. The figure

shows the exact execution time of each query (thin blue line without markers) and the sliding average of 100 queries (green line with circle markers). With the first query, the SMIX initializes its Counter Table and indexes the first value in the Partial Index. With the second query, the SMIX indexes about half of the table into the main memory-based Index Buffer because it has a low Partial Index rate and plenty of buffer space available in the beginning. In consequence, the SMIX execution times of the first two queries (1250 ms and 4880 ms, respectively) exceed the traditional table scan. From the third query on, the execution time of the SMIX is significantly lower than the table scan. When the SMIX has to perform a table scan, it takes about 650 ms to answer the query because the SMIX, still in its unstable state, can leverage the Index Buffer and skip pages during the table scan. Otherwise, if the query hits the Partial Index, the execution time is below 1 ms and therefore comparable to a traditional index. Over the course of the workload, the SMIX collects an increasing share of the queried values in the Partial Index. Consequently, the Partial Index hit rate increases and the average execution time drops quickly below 100 ms.

With query 3196, the Partial Index hit rate exceeds the threshold θ and the SMIX changes into the stable state. Within the process, the SMIX discards its Index Buffer and Counter Table. Without its two helping structures, the SMIX cannot skip pages to speed up the table scan. Consequently, the SMIX execution time in the case of a table scan jumps to 1200 ms. When already well adapted to the current workload, though, the SMIX can answer the majority of queries with a Partial Index scan. The average execution time stays at the low level and drops even further below 50 ms as the SMIX keeps indexing new values in the Partial Index. As is clearly visible in the figure, the decreasing number of spikes indicates that the SMIX has to perform table scans less frequently towards query 5000.

With query 5000, the workload switches to the next episode. None of the values indexed in the SMIX's Partial Index is queried anymore. The Partial Index hit rate drops instantly below the threshold and the SMIX changes into the unstable state again. Within a few queries, the SMIX rebuilds its Index Buffer. The adaptation process repeats.

Figure 5.11 shows the cumulative execution time for the SMIX and the two traditional access paths in comparison. As can be seen, the SMIX quickly becomes worthwhile. With the 10th query, its cumulative execution time falls below the table scan. The bump at query 5000 reflects the cost of re-adaptation. As expected, the traditional full-column index requires less execution time – aside from its creation. In our experimental setting a full-column index on column A requires approximately 107 MB disk space. In comparison, the Partial Index consumes about 3 MB after the first workload episode and 6 MB after the second episode, as illustrated in Figure 5.12. In total, the Partial Index consumes about 10 % of the configured 64 MB available index space. Figure 5.12 shows the amount of memory consumed by the Index Buffer over the course of the workload. Following its nature as an intermediate supporting structure, the Index Buffer consumes all the available buffer space of 64 MB while the SMIX is in the unstable state. In this experiment, the SMIX shows a good adaptation behavior and proves its ability to quickly detect and adapt to workload changes.

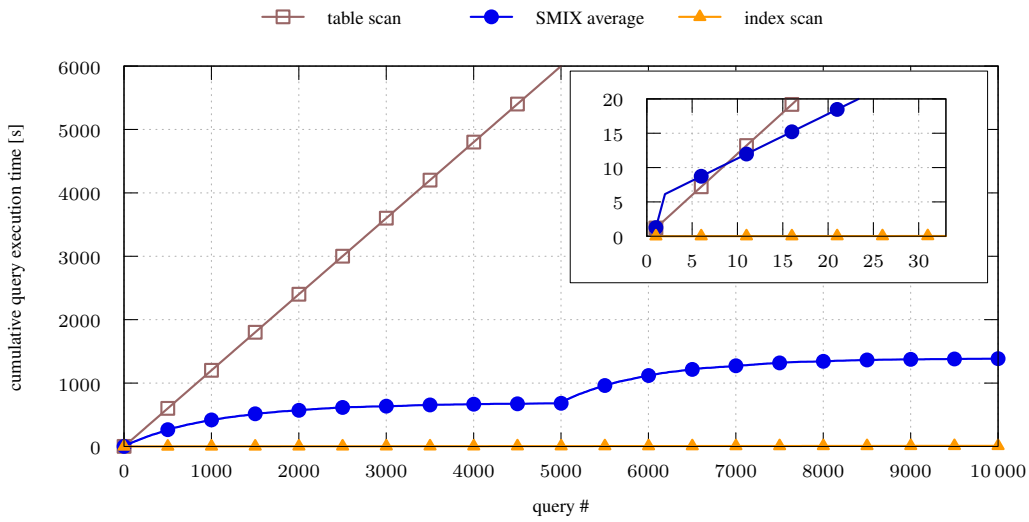


Figure 5.11: Cumulative query execution time.

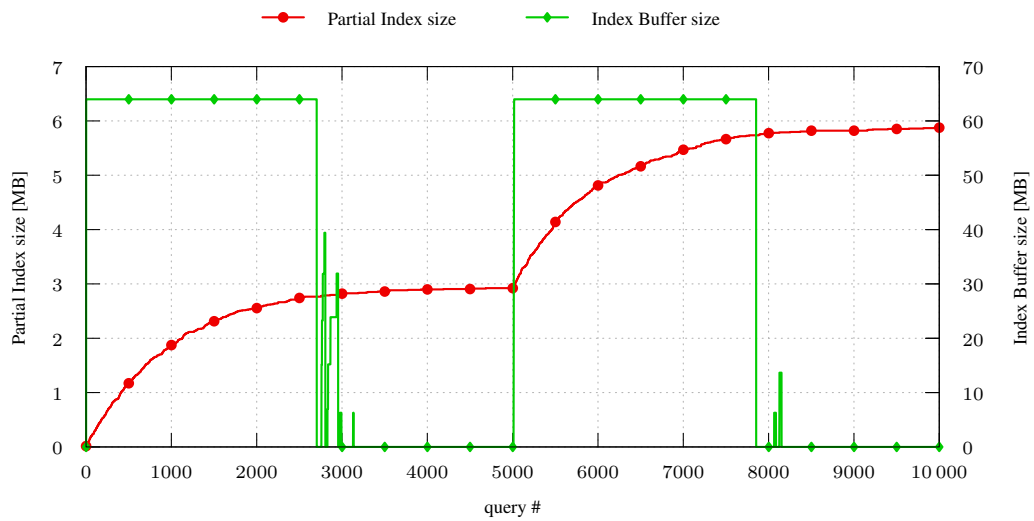


Figure 5.12: Size of Partial Index and Index Buffer.

5.6.3 Parameter Impact

In the next set of experiments we want to investigate the impact of three important SMIX parameters: the *size of the buffer space*, the *stability threshold* θ and the *Partial Index displacement aggressiveness factor* α. All three parameters have influence on the overall adaptation performance. We used the base SMIX configuration (Table 5.5) and the same workload episodes as in the previous experiment. In the following we describe the effects of different settings for each parameter.

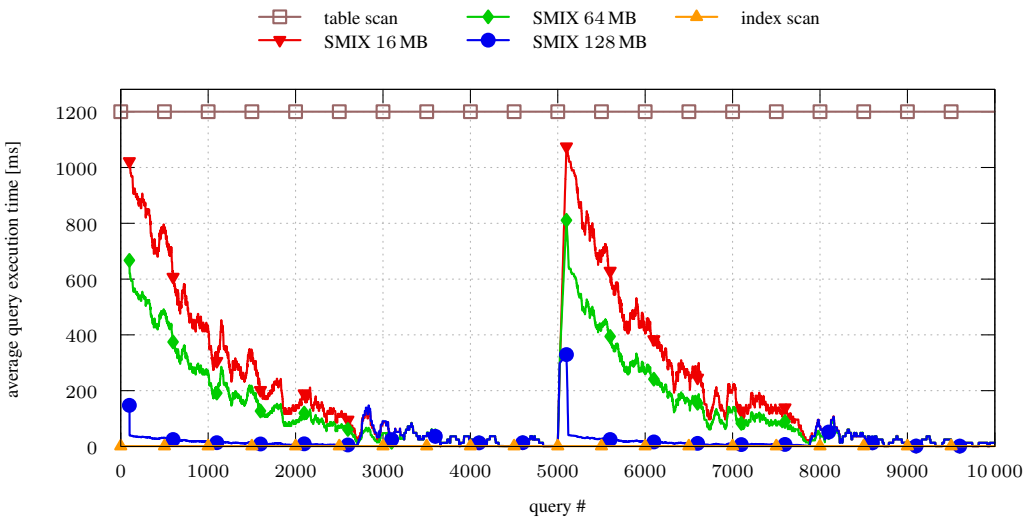


Figure 5.13: Query execution time for different buffer space sizes.

Size of Buffer Space Figure 5.13 shows the execution times (smoothed over a period of 100 queries) for three different sizes of the buffer space. We configured the buffer space with 16 MB, 64 MB, and 128 MB. If Index Buffer indexes all remaining tuples of the relation it would take up to 140 MB. Hence, the buffer space is truely limited in any of the three settings. Because the buffer space is only utilized if a SMIX is in the unstable state, we see big differences in the early adaptation stage. The 128 MB configuration shows the best performance for the first queries of an episode because the Index Buffer is able to index many of the remaining tuples quickly, which allows subsequent table scans to skip most of the pages. At this point, the SMIX operates close to the speed of an index scan. The opposite can be observed at the 16 MB setting. Here, the Index Buffer is able to index merely 1 % of the remaining tuples. This results in a much longer execution time in the early adaptation stage. The often necessary table scans are not able to skip many pages. The larger the buffer space, the better the adaptation behavior. Nevertheless, memory is a costly resource and the buffer space size should be set carefully.

Stability Threshold θ In Figure 5.14 we visualized the smoothed execution times for various settings of the stability threshold θ. We ran the experiments for a threshold of 50 %, 75 %, and 95 %. For this experiment, we see differences as soon as the SMIX enters the stable state. The main observation is that a low threshold of 50 % leads to higher execution times in the early stable stage. This poor performance happens because only 50 % of the queried values hit the Partial Index. Thus, the SMIX needs to invoke a table scan for the other 50 % of the queries, which are not able to skip any pages since the Index Buffer was dropped when the SMIX entered the stable state. In our experiments, a threshold value of 95 % turned out to be the best solution.

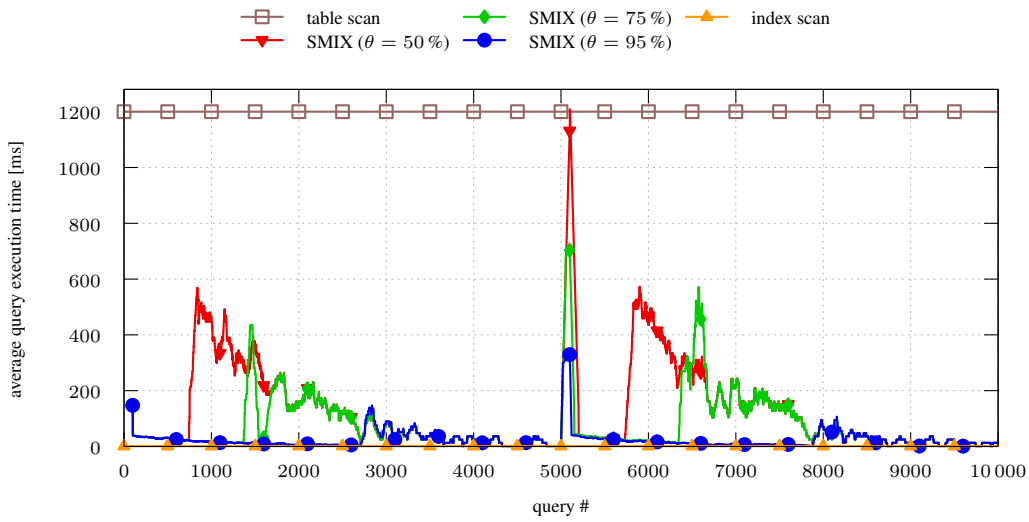


Figure 5.14: Query execution time for different stability thresholds θ.

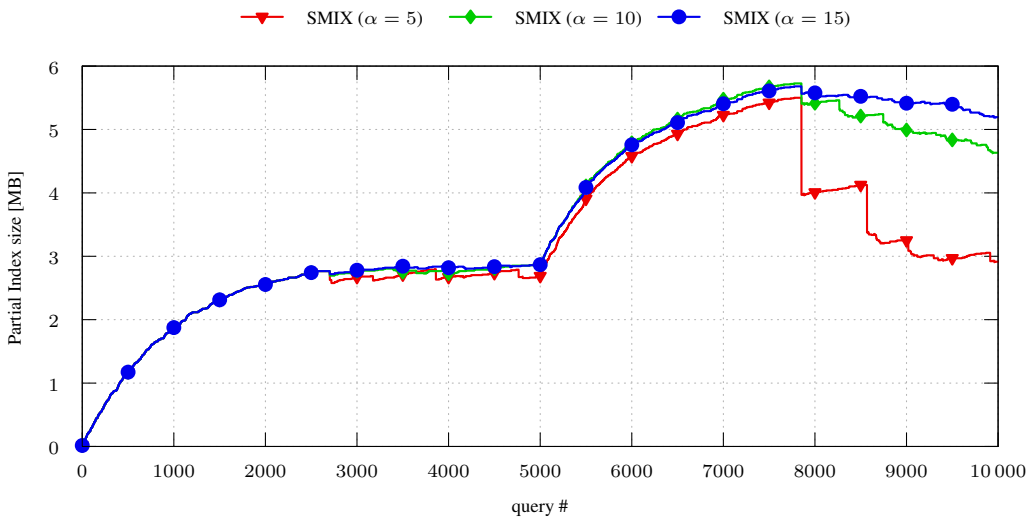


Figure 5.15: Partial Index size for different settings of α.

Partial Index Displacement Aggressiveness α In this experiment we investigated the influence of the automatic displacement, which was disabled in all previous experiments for reasons of simplicity. Because automatic displacement is only allowed in the stable state, we see differences only in the stable stages of the experiment. Figure 5.15 shows the Partial Index size of the SMIX for the settings of 5, 10, and 15. A low α means a high aggressiveness. With α set to 15, we observe a slow displacement of unused Partial Index entries at the end of the second episode. For settings of 10 and 5, the experiment shows a much faster displacement in the second episode.

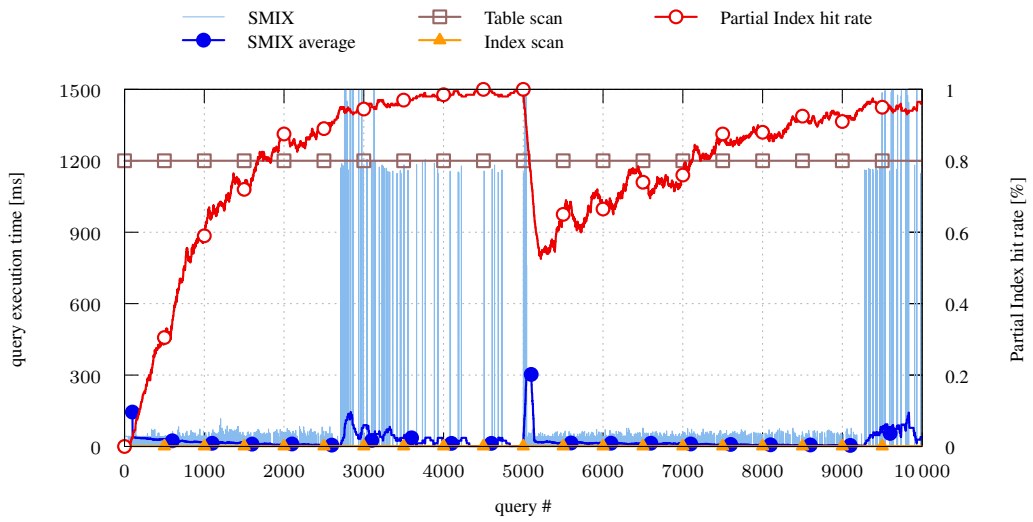


Figure 5.16: Query execution time and Partial Index hit rate on widening workload.

However, the more aggressive displacement also leads to displacements in the first episode, where it affects Partial Index entries that are used by the workload. As a rule of thumb, we recommend $\alpha = 15$ because it is not critical to perform an automatic displacement of stale index information as soon as possible.

5.6.4 Workload Patterns

In this set of experiments, we analyzed the adaptation process for a set of typical workload types. We start with a workload that extends the range of queried values in a single blow. The next workload moves its value range slowly to another position. And finally we investigated a workload that queries a set of scattered values.

Widening Workload In the first episode this workload queries a continuous range of $x \in [1000, 2000]$. The following episode doubles this range to $x \in [1000, 3000]$ and still includes the range of the first episode. Both episodes execute 5000 queries. Figure 5.16 shows the measured execution time and the corresponding Partial Index hit rate. After the first episode the SMIX is well adapted to the workload. As soon as the workload extension happens, the Partial Index hit rate drops below the threshold and ends near 50 % because the Partial Index already indexed half of the value range during the first episode. This falling Partial Index hit rate puts the SMIX into the unstable state, where it uses an Index Buffer to boost the necessary table scans. After query number 9282, the Partial Index hit rate passed the threshold and the SMIX re-enters the stable state and the adaptation to the workload extension is finished.

Shifting Workload This workload consists of three episodes. (1) 3000 queries on a continuous range of $x \in [1000, 2000]$. (2) 6000 queries on a continuous range that

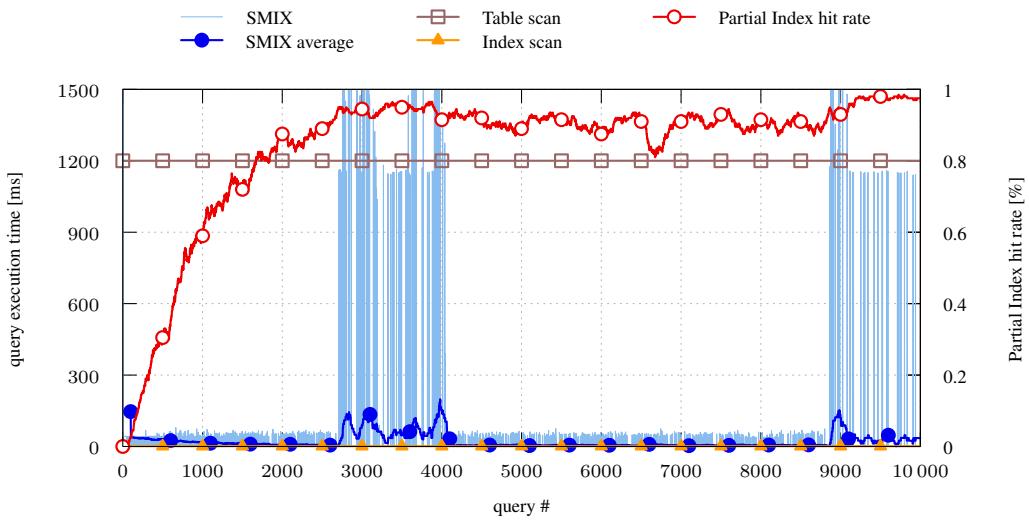


Figure 5.17: Query execution time and Partial Index hit rate on shifting workload.

slides linearly from $x \in [1000, 2000]$ to $x \in [1500, 2500]$. (3) 1000 queries on the final range of the previous episode. Figure 5.17 visualizes execution time and Partial Index hit rate for this workload. At the end of the first episode, the SMIX is well adapted. With the beginning of the next episode, the queried range starts to move slowly. It takes about 1000 queries for the SMIX to detect this slow workload change. After this detection period, the SMIX rebuilds an Index Buffer to support the adaptation process. Once the workload shift is done at the end of the second episode, the SMIX is back in the stable state.

Scattered Workload In many cases, queries are executed on scattered values rather than continuous ranges. Again, the workload consists of two episodes. Each episode consists of 5000 queries on randomly preselected values. Figure 5.18 shows the experimental results. Compared to previous experiments on continuous ranges, we observe a faster rising Partial Index hit rate, but no other visible difference to a workload on a continuous value range.

5.6.5 Complex Scenario

We evaluated the SMIX manager using a more complex workload that involves SMIXs on columns A, B, and C of the table R. The workload executes a total of 15 000 queries and each query has a given probability to hit one of the three columns that changes after 7500 queries. These probabilities are shown in Figure 5.19(a). Furthermore, a query on a specific column addresses a value from a uniformly distributed range of 500 continuous values. We used two disjunct value ranges, where each SMIX starts with the first value range and after a specific number of queries switches over to the second value range. This mapping is visualized in Figure 5.19(b). For instance, query number

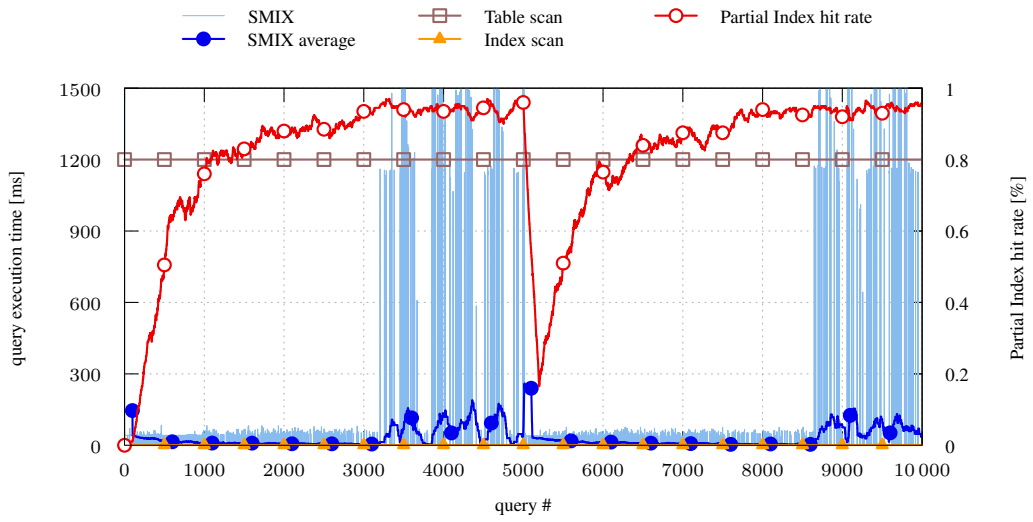
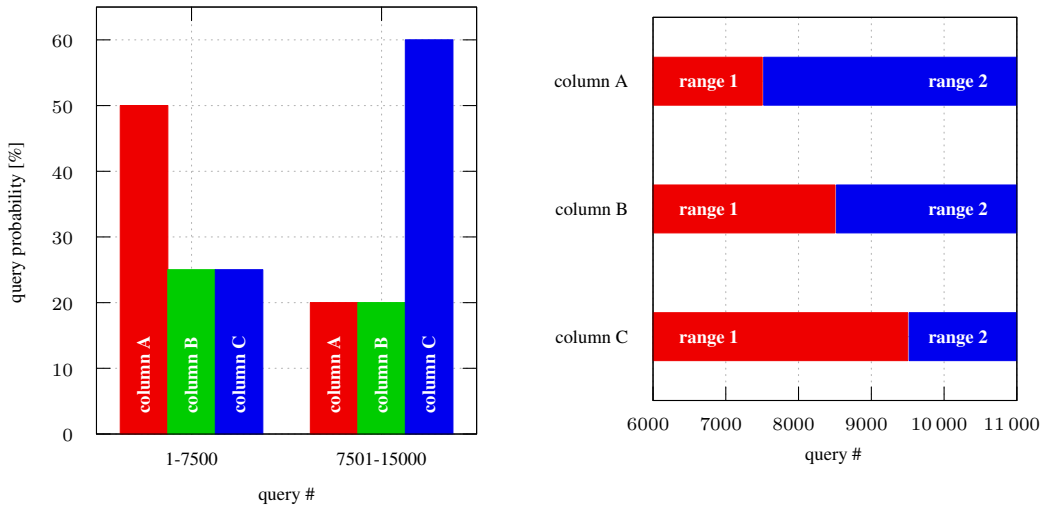


Figure 5.18: Query execution time and Partial Index hit rate on scattered workload.



(a) Query distribution.

(b) Episode switches.

Figure 5.19: Complex scenario setup.

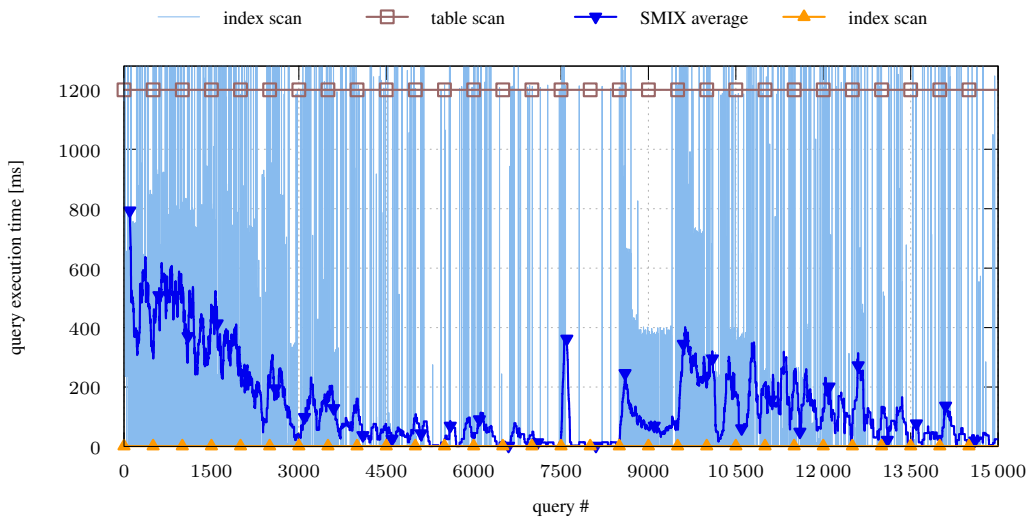


Figure 5.20: Execution time in the complex scenario.

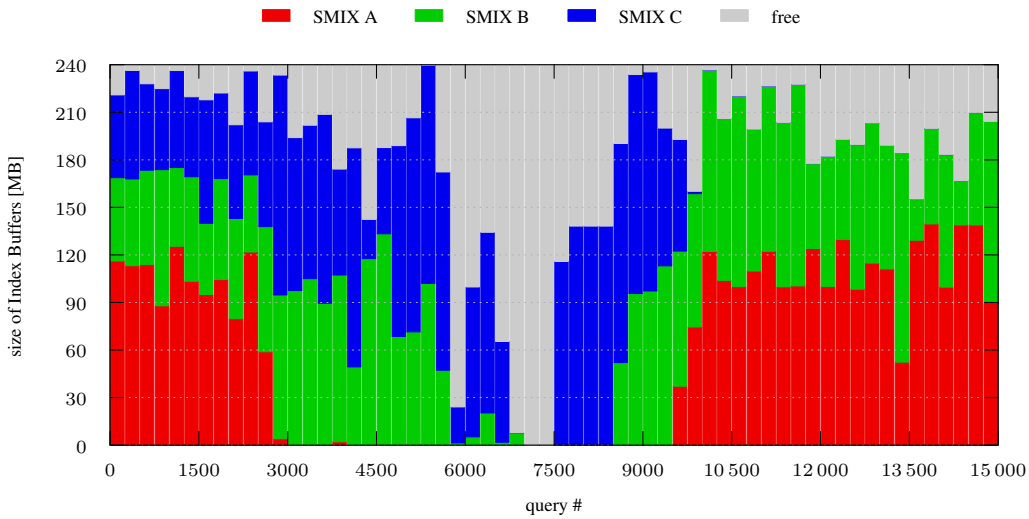


Figure 5.21: Buffer space occupancy in the complex scenario.

8000 has a probability of 60 % to hit $SMIX_C$, and when that occurs x is a random value in the first value range. For the experiment, we set the size of the buffer space to 256 MB. The execution time (averaged over a period of 100 queries) is shown in Figure 5.20. Figure 5.21 shows the corresponding size of the Index Buffer for all three SMIXs. At the beginning, all SMIXs start in the unstable state. For that reason, every SMIX builds up an Index Buffer to speed up table scans. Because of the limited buffer space, all three SMIXs have to compete for the available memory. Therefore, the SMIX manager assigns a buffer space share to each SMIX. This share mainly

depends on the access frequency of a SMIX. Consequently, $SMIX_A$ receives twice as much buffer space than $SMIX_B$ and $SMIX_C$. After query number 2725 $SMIX_A$ is mostly in the stable state and not involved in the buffer space distribution anymore. Thus, $SMIX_B$ and $SMIX_C$ are able to grow and speed up their table scans until all SMIXs are finally stable after query number 6899. With query 7501, the column access distribution changes and the value range for column C changes. Therefore, this SMIX builds up its Index Buffer again. It does not have to compete with other SMIXs since all of them are stable. 1000 queries later, $SMIX_B$ also changes its value range and starts to compete with $SMIX_C$. Finally, $SMIX_A$ joins this competition, too. Shortly afterwards $SMIX_C$ becomes stable again, so that only $SMIX_A$ and $SMIX_B$ are left to compete for buffer space.

5.6.6 Partitioned Index Buffer

The evaluation of the partitioned Index Buffer was done in a different setup. We implemented the Index Buffer concept prototypically in the H2 Database Engine 1.3 [Mueller, 2012]. We ran all experiments on a machine with an Intel Core 2 Duo processor at 1.6 GHz, 4 GB of DDR3 main memory, and a 128 GB Samsung SSD. We used Microsoft Windows 7 64 bit edition as the operating system and Java SE 6 as the runtime environment.

For all experiments on the partitioned Index Buffer, we again used a single table with three INTEGER columns A, B, and C for indexing and one VARCHAR(512) column as payload. The integer columns are populated with random values uniformly distributed from 1 to 50 000. The size of the payload values is also uniformly distributed, but ranges from 1 to 512 bytes. We filled the table with 500 000 tuples, resulting in an effective table size of 220 MB on disk. To isolate the effects of the partitioned Index Buffer, we turned off the other SMIX features. In each column, the Partial Index indexes a fixed subset of values, specifically the top 10 % of the value range, i.e. values from 1 to 5000. In experiments 1, 2, and 3 we queried the unindexed values randomly. Experiment 4 shows the case where queries also address values covered by the Partial Index. In each experiment the workload consists of 200 queries.

The first experiment illustrates the basic behavior of a single Index Buffer. Accordingly, we queried only column A. The buffer space was set to unlimited size, $I^{\text{MAX}} = 5000$ pages were indexed at most during a table scan, and each Index Buffer partition indexed a maximum of $P = 10\,000$ pages. For each query, we measured the execution time of individual queries, the total number of entries in the Index Buffer, and the number of pages that were skipped. Figure 5.22 shows the results. For comparison, the figure also shows the execution time of the same queries without the Index Buffer. As can be seen, the first couple of queries exhibit a slightly high execution time, but quickly the execution time drops below the level of the table scan. The obvious reason is the Index Buffer, which indexed an increasing number of tuples. The table scans are quickly able to skip a large number of pages. Since the buffer space is of unlimited size in this experiment, all pages are completely indexed after 20 queries. At that point, the query execution time is the same at that of an index scan.

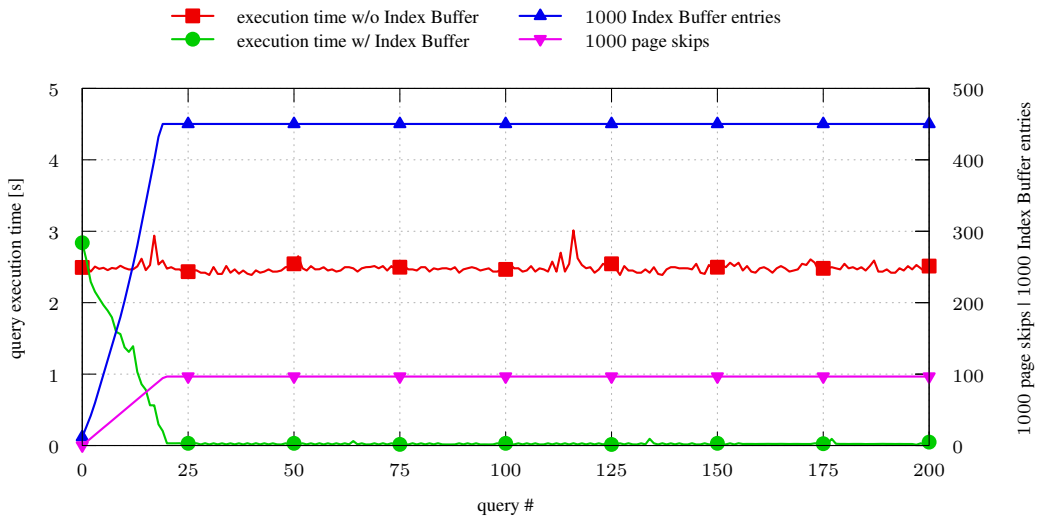


Figure 5.22: Query execution time with Index Buffer ($I^{\text{MAX}} = 5000$).

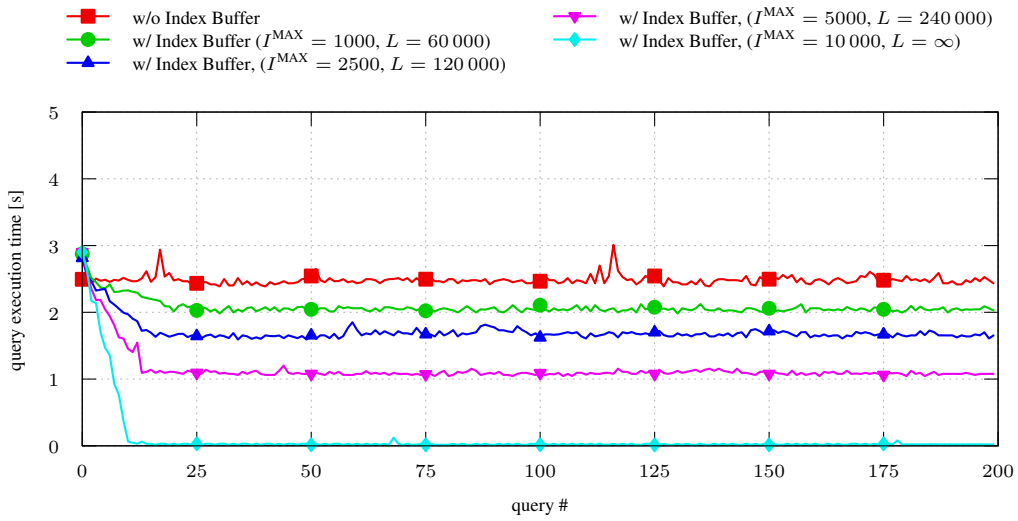


Figure 5.23: Query execution time with Index Buffer for different buffer space sizes and I^{MAX}.

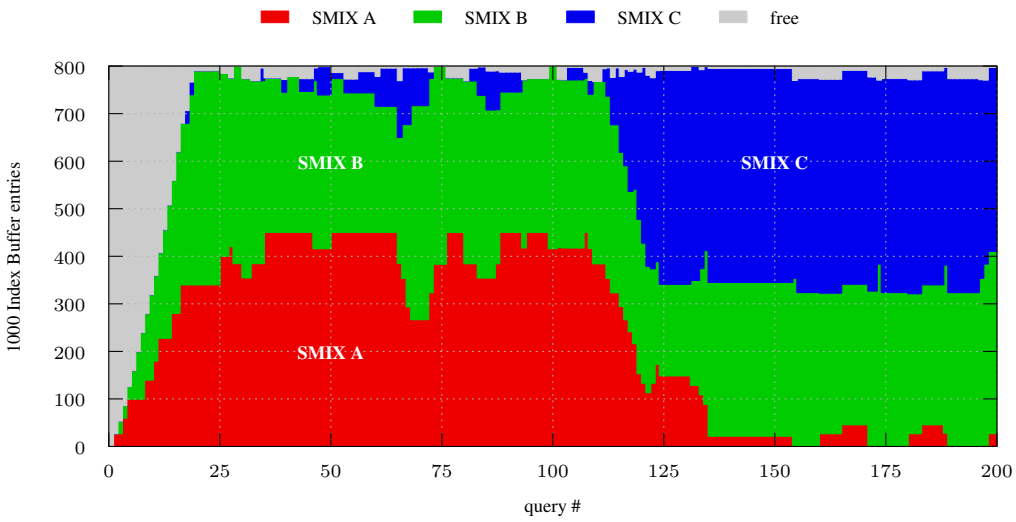


Figure 5.24: Size of three Index Buffers with limited space.

The second experiment shows the influence of the maximum number of pages indexed per table scan I^{MAX} and the space bound of the buffer space L. We ran this experiment with the same setting as the first experiment, except that we varied I^{MAX} and L. The effects of I^{MAX} and L are independent from each other. As Figure 5.23 shows, I^{MAX} determines how aggressively the Index Buffer indexes new pages. The higher I^{MAX}, the more pages are indexed during one scan. In consequence, the query execution time drops more quickly within the first 15 queries with higher I^{MAX}. Of course, the Index Buffer also occupies buffer space more quickly with higher I^{MAX} (not shown in the figure). The size of the buffer space limits the maximum number of entries and with it the number of pages that can be skipped. As Figure 5.23 also shows, the smaller the buffer space, the less the Index Buffer can speed up table scans.

The third experiment shows the Index Buffer management. We ran the workload of 200 queries on all three columns. Half of the queries select tuples on column A, one third on column B, and one sixth on column C. After 100 queries, we switched the query mix to: One sixth on column A, one third on column B, and one half on column C. The buffer space was limited to 800 000 entries, at most $I^{\text{MAX}} = 5000$ pages were indexed during each table scan, and each Index Buffer partition indexed a maximum of $P = 10\,000$ pages. Figure 5.24 shows the number of entries in each of the three Index Buffers. In the first workload period the Index Buffer on SMIX A occupies more than half of the buffer space. The Index Buffer on SMIX B occupies most of the remaining buffer space, while the Index Buffer on SMIX C only sporadically accumulates entries. After the change of the query mix, the situation quickly turns around. In the second workload period, the Index Buffer on SMIX C rapidly grows to roughly 55 % of the buffer space and the Index Buffer on SMIX A shrinks practically to zero.

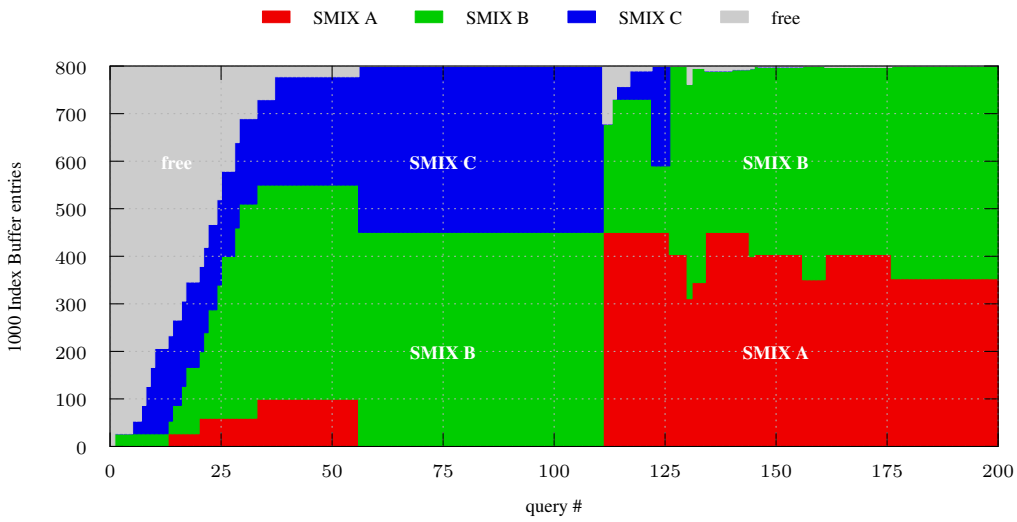


Figure 5.25: Size of three Index Buffers with limited space and Partial Index hits on the SMIX of column A.

The fourth experiment considers the Index Buffer management under the influence of different Partial Index hit rates. The general setting is similar to the third experiment except that the queries on column A also query values covered in the Partial Index on that column. To show the influence of the Partial Index hits on the allocation of buffer space, we switch the set of values indexed in the Partial Index after 100 queries. Among the first 100 queries, queries on column A hit the Partial Index with a probability of 80 %. During the following 100 queries the partial index hit rate for column A queries is only 20 %. The query distribution is fixed during the complete workload. Specifically, half of the queries run against column A, one third against column B, and one sixth against column C. The buffer space settings were the same as in the third experiment. The maximum number of pages to index in one table scan is $I^{\text{MAX}} = 10\,000$ and the maximum number of pages an Index Buffer partition can index was set to $P = 10\,000$. Figure 5.25 shows the three Index Buffers and their entries in the buffer space. After the buffer space is filled the Index Buffers competed for space. For the first period the Partial Index of SMIX A was hit frequently and the SMIX manager decided to give less space to this Index Buffer. Thus, SMIX B and SMIX C gain more buffer space, though they are queried less often. This state changes after 100 queries and the Index Buffer of SMIX A is used more often than in the first period. As can be seen, the Index Buffer of SMIX A gets more buffer space and grows quickly after the Partial Index hit rate changes, whereas the Index Buffers of SMIX B and SMIX C shrink.

5.7 Summary

With SMIX we have proposed a novel, adaptable, fine-grained, autonomous indexing infrastructure. It builds on a novel default access path, also called SMIX, which combines the traditional table scan with index structures. A SMIX exhibits two key features: First, it automatically indexes the most frequently queried values completely, and additionally completes indexing of pages during periods of workload adaptation to lower the cost of necessary table scans. Second, it is able to discard index entries if they become less useful. With these two features, SMIXs continuously adapt to the database workload and the database schema and control their resource usage at the same time. All SMIXs within a SMIX population compete for resources; the most frequently used SMIX gets the most resources. The SMIX manager component supervises this competition by continuously determining resource quotas for every SMIX depending on its usage. The partitioned Index Buffer is an extension to the SMIX concept that allows an even more subtle management of the resources used for indexing. In a series of experiments, we evaluated how SMIXs operate and perform. SMIXs showed significantly better performance than traditional scans. In periods of constant workload, SMIXs reach the same performance plateau as traditional indexes, while requiring fewer storage resources if the workload is focused. During periods of changing workloads or evolving schemas, SMIXs consume additional memory resources to boost query performance. We strongly believe that the SMIX approach points into a new direction of how we can build the autonomous indexing infrastructures of the future, to lower the total cost of indexing in FRDM databases.

6 Conclusions

An increasing number of application areas of data management technology demands flexible and agile schema-comes-second data management. Agile analytics and agile software development are just two examples. Driven by the restlessness of the Web and mobile applications, software in general is considered an agile and vivid artifact constantly in flux. Relational database systems are by far the most successful, popular, and mature data management technology. Their design, however, originates from the 1970s when schema-comes-first was common and appropriate for the database applications at the time. This led to the prescriptive nature relational database systems still exhibit today. Since prescriptive schema-comes-first thinking is unsuitable in many application areas today and simply not in vogue anymore, relational database technology is increasingly perceived as anachronistic. Striving for more flexibility, developers often favor so called NoSQL systems. Although NoSQL systems throw off the chains of schema-comes-first thinking, they also give up many of the good principles of database technology that have been established in over thirty years of research and development in the field.

In this thesis, we took an evolutionary approach for flexible database management systems. We tried to build on the mature and proven technology of relational database management systems and make them ready for the flexibility- and agility-demanding schema-comes-second data management of today. In adapting relational database technology towards descriptive schema-comes-second data management, we aimed at bridging the gap between the persistent prevalence of relational technology in organizations and its decreasing reputation among developers. In the thesis we presented four technological concepts that contribute to this goal:

FRDM – A Flexible Relational Data Model FRDM is descriptive and free of implicit constraints. It allows the representation of entities with irregular attribute sets and logical domain memberships. Attributes and logical entity domains are independent from each other, as are attributes and technical types. All this gives FRDM the flexibility necessary for schema-comes-second data management. Where more regularly structured data is dominant and the database management should help to establish and ensure data quality, the constraint framework FRDM-C explicitly allows restricting FRDM's flexibility. FRDM combines its flexible data representation with the processing power of the relational algebra. FRDM is compatible and interoperable with relational data.

ADOM – Autonomous Physical Entity Domains ADOM is a means to avoid storing irregularly structured schema-comes-second data in a universal physical

domain. ADOM continuously and autonomously partitions irregularly structured entities into multiple homogenous physical entity domains. The partitioning is based on the entities' schema properties. This allows queries to prune physical entity domains of irrelevant entities before touching the actual data. Compared to a universal physical entity domain, ADOM helps to increase retrieval performance on irregularly structured data, as we were able to show experimentally. ADOM works online and is designed for low overhead. We have published a paper on ADOM in an early version in Herrmann et al. [2014] and, in the version of the thesis, got accepted for publication on the 9th International Workshop on Self-Managing Database Systems 2014.

FASE – A Freely Adjustable Storage Engine FASE allows configuring of the physical data layout with the help of the FASE notation. The FASE notation is a description language to specify the macroscopic characteristics of physical data layouts, i.e. how data elements are grouped on the physical storage layer. For schema-comes-second database systems FASE is particularly useful. On the one hand, irregularly structured data requires a different physical data layout than regularly structured data. On the other hand, the degree of irregularity is very variable. With FASE, the physical data layout is not hard-coded but part of the physical design, and can be adjusted to the irregularity properties of an evolving data set, and to the requirements of a changing workload. FASE increases the physical data independence of database systems.

SMIX – Self-managed Indexing SMIX takes the DBA out of the loop of index tuning. In schema-comes-second databases, attributes available in a database are not known in advance and change with the data. The schema and workload are both dynamic in many of today's databases. Optimizing the configuration of secondary indexes in such databases becomes a constant challenge. SMIX autonomously indexes the most queried tuples in a database and keeps the amount of secondary index information in a given resource bound. To achieve this, SMIX is built on a novel access path, which combines the traditional tables scan and the traditional index scan. When used, the access path collects index information to speed up subsequent retrievals. To stay within the given resource limits, SMIX discards infrequently used index information. By these means, SMIX can continuously adapt the set of index information in a database to dynamic workloads, as well as to dynamic schemas. We have published a paper on SMIX internationally in Voigt et al. [2013], Kissinger et al. [2012], and Voigt et al. [2012].

We are confident that these four contributions help pave the way to a more flexible future for relational database management technology.

Future Work

This thesis can serve as a starting point for further interesting research on the flexibility and manageability aspects of data management. In the following, we sketch three possible future projects.

NF^2 Semantics in FRDM Compared to many other data models currently in vogue in investigative analytics and web application development, FRDM lacks hierarchical structures. We argued that hierarchical structures are merely a limited way of referencing. From a pure data representation perspective, hierarchical referencing is redundant if a data model offers more powerful and more flexible referencing mechanisms, such as value-based referencing. Nevertheless, hierarchical structures gained a certain popularity, because they are very handy when the entities and facts that should be represented are of inherently hierarchical nature. Examples are event data, document-like data, and CAD and GIS data. Of a very similar nature are so-called multi-valued attributes, i.e. where lists or sets of values are allowed as instances of value domains. Multi-valued attributes do not increase expressiveness of a data model such as FRDM but make it simpler to represent certain information. Examples of multiple values belonging to a single attribute are a person's hobbies or the supported connectivity standards of a smartphone. Non-First Normal Form (NF^2) relations already introduced multi-valued attributes and hierarchical structures to the relational data model in the 1980s [Schek and Pistor, 1982, Jaeschke and Schek, 1982, Pistor and Andersen, 1986]. Because of its relational nature, the basic concept of NF^2 is similarly applicable to FRDM. However, NF^2 was developed for the prescriptive schema-comes-first world. Its concepts have to be adapted to the descriptive nature of schema-comes-second data management to be of any value in FRDM. This raises many questions, such as: How much schema flexibility is allowed for entities nested under the same attribute? How are nested entities of variable schema handled in queries? Introducing NF^2 semantics to FRDM is not well understood yet, but a field well worth of being researched.

A Common Database Programming and Execution Layer Although the relational data model is still the dominant data model in the field of data management, other data models are gaining ground. In particular, graph models or document models are very popular in certain application domains. Some relational systems offer support for models such as XML [Cheng and Xu, 2000, Schmidt et al., 2001, Nicola and der Linden, 2005, Acharya et al., 2008]) or JSON [Monash, 2013b]. SAP goes a step further. SAP HANA integrates multiple engines for relational data, text data [Färber et al., 2011, 2012], graph data [Rudolf et al., 2013], and matrix data [Kernert et al., 2013]. All these engines are based on the same main-memory column store technology, i.e. they share the same storage layout. Another trend in database technologies is engines that support multiple storage layouts, as we have described in Chapter 4. It would

be worthwhile to allow both multiple models and multiple data layouts in a single engine. To make architectural sense, such a diverse database engine needs a common architectural layer, which serves as the core programming and execution environment [Habich et al., 2011]. FASE can be the nucleus of such a common layer. As presented, FASE is built around the following four domains: entity types T, entities E, attributes A and values V, which originate from the relational model. Using other domains would allow the natural representation of other forms of data. For instance, the four domains nodes, edges, attributes, and values could serve as a basis for graph data, while the three domains x, y, and values are sufficient to represent matrix data. The operations FASE offers work on any such domain sets. Specific engines on top realize the complete support of a given data model. Nevertheless, these specific engines can share the powerful functionality and the adaptable data layout of FASE. While this is an appealing vision, its realization requires additional research. Particularly, data models and their typical operation sets have to be studied to understand what features they have in common and how they can be generalized. FASE can serve as a good starting point for such an endeavor.

Self-managed Storage Engine Another interesting research direction that emanates from FASE is the field of self-managed data stores. Although FASE and other hybrid data stores support various data layouts, they still require a DBA to configure the physical data layout. The only work we are aware of that points in the direction of automating this is HYRISE [Grund et al., 2010]. HYRISE features a design advisor to recommend optimal column groups for a given workload. However, it is desirable for future data stores to autonomously choose the data layouts best suited for a database. This should also involve mixed data layouts. Additionally, the data store should adapt the layout to workload changes over time. On the way to such a self-managed storage engine various problems have to be investigated and solved, for instance: How can we model the access characteristics that favor certain data layouts, and how can these characteristics be detected in a workload? How can a running system incrementally and efficiently change the physical data layout without impairing the ongoing query processing? What is a reasonable granularity for mixed layouts, which offers high adaptability but still allows exploiting of the positive characteristics of a particular layout?

Bibliography

Daniel J. Abadi, Adam Marcus, Samuel Madden, and Katherine J. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *VLDB'07, Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 411–422. ACM, 2007. ISBN 978-1-59593-649-3.

Daniel J. Abadi, Samuel Madden, and Nabil Hachem. ColumnStores vs. RowStores: How Different Are They Really? In Jason Tsong-Li Wang, editor, *SIGMOD'08, Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 967–980. ACM, 2008. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376712.

Pekka Abrahamsson, Juhani Warsta, Mikko T. Siponen, and Jussi Ronkainen. New Directions on Agile Methods: A Comparative Analysis. In Lori A. Clarke, Laurie Dillon, and Walter F. Tichy, editors, *ICSE'03, Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*, pages 244–254. IEEE Computer Society, 2003. doi: 10.1109/ICSE.2003.1201204.

Srini Acharya, Peter Carlin, César A. Galindo-Legaria, Krzysztof Kozielczyk, Pawel Terlecki, and Peter Zabback. Relational support for flexible schema scenarios. *The Proceedings of the VLDB Endowment*, 1(2):1289–1300, 2008.

Rakesh Agrawal and Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 487–499. Morgan Kaufmann, 1994.

Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining Association Rules between Sets of Items in Large Databases. In Peter Buneman and Sushil Jajodia, editors, *SIGMOD'93, Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 207–216. ACM Press, 1993. doi: 10.1145/170035.170072.

Rakesh Agrawal, Amit Somani, and Yirong Xu. Storage and Querying of E-Commerce Data. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB'01, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 149–158. Morgan Kaufmann, 2001. ISBN 1-55860-804-4.

Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB'00, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 496–505. Morgan Kaufmann, 2000. ISBN 1-55860-715-3.

Sanjay Agrawal, Vivek R. Narasayya, and Beverly Yang. Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors, *SIGMOD'04, Proceedings of the 2004 ACM SIGMOD International*

Conference on Management of Data, Paris, France, June 13-18, 2004, pages 359–370. ACM, 2004. ISBN 1-58113-859-8. doi: 10.1145/1007568.1007609.

Sanjay Agrawal, Eric Chu, and Vivek R. Narasayya. Automatic Physical Design Tuning: Workload as a Sequence. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *SIGMOD'06, Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 683–694. ACM, 2006. ISBN 1-59593-256-9. doi: 10.1145/1142473.1142549.

Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving Relations for Cache Performance. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB'01, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 169–180, 2001. ISBN 1-55860-804-4.

David J. Anderson. *Agile Management for Software Engineering: Applying the Theory of Constraints for Business Results*. Prentice Hall, September 2003. ISBN 0-13-142460-2.

David J. Anderson. *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press, Sequim, Washington, April 2010. ISBN 0-9845214-0-2.

ANSI/X3/SPARC. Interim Report: ANSI/X3/SPARC Study Group on Data Base Management Systems 75-02-08. *Bulletin of ACM SIGMOD*, 7(2):1–140, 1975.

Apache Software Foundation. *The Apache HBase^{TM} Reference Guide*, chapter 5: Data Model. Apache Software Foundation, 2013.

Deborah J. Armstrong. The quarks of object-oriented development. *Communications of the ACM*, 49 (2):123–128, 2006. doi: 10.1145/1113034.1113040.

Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary G. Ives. DBpedia: A Nucleus for a Web of Open Data. In Karl Aberer, Key-Sun Choi, Natasha Fridman Noy, Dean Allemang, Kyung-Il Lee, Lyndon J. B. Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Riichiro Mizoguchi, Guus Schreiber, and Philippe Cudré-Mauroux, editors, *ISWC/ASWC'07, The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007*, volume 4825 of *Lecture Notes in Computer Science*, pages 722–735. Springer, 2007. ISBN 978-3-540-76297-3. doi: 10.1007/978-3-540-76298-0_52.

Stefan Aulbach, Michael Seibold, Dean Jacobs, and Alfons Kemper. Extensibility and Data Sharing in evolving multi-tenant databases. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *ICDE'11, Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 99–110. IEEE Computer Society, 2011. ISBN 978-1-4244-8958-9. doi: 10.1109/ICDE.2011.5767872.

David E. Avison and Guy Fitzgerald. Where Now for Development Methodologies? *Communications of the ACM*, 46(1):79–82, 2003. doi: 10.1145/602421.602423.

Don S. Batory. Modeling the Storage Architectures of Commercial Database Systems. *ACM Transactions on Database Systems*, 10(4):463–528, 1985. doi: 10.1145/4879.5392.

Don S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise. GENESIS: An Extensible Database Management System. *IEEE Transactions on Software Engineering*, 14(11):1711–1730, 1988. doi: 10.1109/32.9057.

Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1:173–189, 1972. doi: 10.1007/BF00288683.

Kent Beck. Embracing Change with Extreme Programming. *IEEE Computer*, 32(10):70–77, 1999. doi: 10.1109/2.796139.

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for Agile Software Development. `http://agilemanifesto.org/`, 2001.

Jennifer L. Beckmann. The CNET E-Commerce Data Set. Technical report, University of Wisconsin, Madison, July 2005.

Jennifer L. Beckmann, Alan Halverson, Rajasekar Krishnamurthy, and Jeffrey F. Naughton. Extending RDBMSs To Support Sparse Datasets Using An Interpreted Attribute Storage Format. In Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *ICDE'06, Proceedings of the 22nd International Conference on Data Engineering, 3-8 April 2006, Atlanta, GA, USA*, page 58. IEEE Computer Society, 2006. doi: 10.1109/ICDE.2006.67.

Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, pages 34–43, May 2001.

Tim Berners-Lee, Roy Goldfinger Fielding, and Larry Masinter. Uniform Resource Identifier (URI): Generic Syntax, RFC 3986. `http://tools.ietf.org/html/rfc3986`, January 2005.

Philip A. Bernstein. Synthesizing Third Normal Form Relations from Functional Dependencies. *ACM Transactions on Database Systems*, 1(4):277–298, 1976. doi: 10.1145/320493.320489.

Kurt D. Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: A Collaboratively Created Graph Database For Structuring Human Knowledge. In Jason Tsong-Li Wang, editor, *SIGMOD'08, Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, Vancouver, British Columbia, Canada, June 10-12, 2008.*, pages 1247–1250, 2008. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376746.

Peter A. Boncz and Martin L. Kersten. MIL Primitives for Querying a Fragmented World. *The VLDB Journal – The International Journal on Very Large Data Bases*, 8(2):101–119, 1999. doi: 10.1007/s007780050076.

R. Bonnano, Dario Maio, and Paolo Tiberio. An Approximation Algorithm for Secondary Index Selection in Relational Database Physical Design. *The Computer Journal*, 28(4):398–405, 1985. doi: 10.1093/comjnl/28.4.398.

Robert Bosak, Richard F. Clippinger, Carey Dobbs, Roy Goldfinger, Renee B. Jasper, William Keating, George Kendrick, and Jean E. Sammet. An Information Algebra: Phase 1 Report – Language Structure Group of the CODASYL Development Committee. *Communications of the ACM*, 5(4):190–204, April 1962. doi: 10.1145/366920.366935.

Gilad Bracha and William R. Cook. Mixin-based Inheritance. In Akinori Yonezawa, editor, *OOPSLA/ECOOP'90, Conference on Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming (OOPSLA/ECOOP), Ottawa, Canada, October 21-25, 1990, Proceedings*, pages 303–311. ACM, 1990. ISBN 0-89791-411-2. doi: 10.1145/97945.97982.

Michael L. Brodie and Jason T. Liu. OTM'10 Keynote: The Power and Limits of Relational Technology In the Age of Information Ecosystems. In *OTM'10, On the Move to Meaningful Internet Systems: Confederated International Conferences, Hersonissos, Crete, Greece, October 25-29, 2010, Proceedings*, volume 6426 of *Lecture Notes in Computer Science*, pages 2–3. Springer, 2010. ISBN 978-3-642-16933-5. doi: 10.1007/978-3-642-16934-2_2.

Michael L. Brodie and Joachim W. Schmidt. Final Report of the ANSI/X3/SPARC DBS-SG Relational Database Task Group. *SIGMOD Record*, 12(4):i–62, 1982.

Nicolas Bruno. Teaching an Old Elephant New Tricks. In *CIDR'09, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*, 2009.

Nicolas Bruno and Surajit Chaudhuri. Automatic Physical Database Tuning: A Relaxation-based Approach. In Fatma Özcan, editor, *SIGMOD'05, Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 227–238. ACM, 2005. ISBN 1-59593-060-4. doi: 10.1145/1066157.1066184.

Nicolas Bruno and Surajit Chaudhuri. Physical Design Refinement: The "Merge-Reduce" Approach. In Yannis E. Ioannidis, Marc H. Scholl, Joachim W. Schmidt, Florian Matthes, Michael Hatzopoulos, Klemens Böhm, Alfons Kemper, Torsten Grust, and Christian Böhm, editors, *EDBT'06, 10th International Conference on Extending Database Technology, Munich, Germany, March 26-31, 2006, Proceedings*, volume 3896 of *Lecture Notes in Computer Science*, pages 386–404. Springer, 2006a. ISBN 3-540-32960-9. doi: 10.1007/11687238_25.

Nicolas Bruno and Surajit Chaudhuri. To Tune or not to Tune? A Lightweight Physical Design Alerter. In Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors, *VLDB'06, Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 499–510. ACM, 2006b. ISBN 1-59593-385-9.

Nicolas Bruno and Surajit Chaudhuri. An Online Approach to Physical Design Tuning. In *ICDE'07, Proceedings of the 23nd International Conference on Data Engineering, April 15-20, 2007, The Marmara Hotel, Istanbul, Turkey*, pages 826–835. IEEE, 2007a. doi: 10.1109/ICDE.2007.367928.

Nicolas Bruno and Surajit Chaudhuri. Physical design refinement: The "merge-reduce" approach. *ACM Transactions on Database Systems*, 32(4), 2007b. doi: 10.1145/1292609.1292618.

Nicolas Bruno and Rimma V. Nehme. Configuration-Parametric Query Optimization for Physical Design Tuning. In Jason Tsong-Li Wang, editor, *SIGMOD'08, Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, Vancouver, British Columbia, Canada, June 10-12, 2008.*, pages 941–952. ACM, 2008. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376710.

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*, 7(6):955–967, 1995.

Ümit V. Çatalyürek and Cevdet Aykanat. Decomposing Irregularly Sparse Matrices for Parallel Matrix-Vector Multiplication. In *IRREGULAR'96, Parallel Algorithms for Irregularly Structured Problems, Third International Workshop, Santa Barbara, California, USA, August 19-21, 1996, Proceedings*, volume 1117 of *Lecture Notes in Computer Science*, pages 75–86. Springer, 1996. ISBN 3-540-61549-0. doi: 10.1007/BFb0030098.

Ümit V. Çatalyürek and Cevdet Aykanat. Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10 (7):673–693, 1999. doi: 10.1109/71.780863.

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI'06, 7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 205–218. USENIX Association, 2006.

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26(2), 2008. doi: 10.1145/1365815.1365816.

Kevin Chen-Chuan Chang, Bin He, Chengkai Li, Mitesh Patel, and Zhen Zhang. Structured Databases on the Web: Observations and Implications. *SIGMOD Record*, 33(3):61–70, 2004. doi: 10.1145/1031570.1031584.

Surajit Chaudhuri and Vivek R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 146–155. Morgan Kaufmann, 1997. ISBN 1-55860-470-7.

Surajit Chaudhuri and Vivek R. Narasayya. AutoAdmin 'What-if' Index Analysis Utility. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD'98, Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, Seattle, Washington, USA, June 2-4, 1998*, pages 367–378. ACM Press, 1998. ISBN 0-89791-995-5. doi: 10.1145/276304.276337.

Surajit Chaudhuri and Vivek R. Narasayya. Index Merging. In *ICDE'99, Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Austrialia*, pages 296–303. IEEE Computer Society, 1999. doi: 10.1109/ICDE.1999.754945.

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*, 16(11):1313–1323, 2004. doi: 10.1109/TKDE.2004.75.

Peter P. Chen. The Enity-Relationship Model: Toward a Unified View of Data. In Douglas S. Kerr, editor, *VLDB'75, Proceedings of the International Conference on Very Large Data Bases, September 22-24, 1975, Framingham, Massachusetts, USA*, page 173. ACM, 1975.

Josephine M. Cheng and Jane Xu. XML and DB2. In *ICDE'00, Proceedings of the 16th International Conference on Data Engineering, 28 February - 3 March, 2000, San Diego, California, USA*, pages 569–573. IEEE Computer Society, 2000. doi: 10.1109/ICDE.2000.839455.

Sunil Choenni, Henk M. Blanken, and Thiel Chang. Index Selection in Relational Databases. In Osman Abou-Rabia, Carl K. Chang, and Waldemar W. Koczkodaj, editors, *ICCI'93, Fifth International Conference on Computing and Information, Sudbury, Ontario, Canada, May 27-29, 1993, Proceedings*, pages 491–496. IEEE Computer Society, 1993. ISBN 0-8186-4212-2.

Eric Chu, Jennifer L. Beckmann, and Jeffrey F. Naughton. The Case for a Wide-Table Approach to Manage Sparse Relational Data Sets. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *SIGMOD'07, Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 821–832. ACM, 2007. ISBN 978-1-59593-686-8. doi: 10.1145/1247480.1247571.

Alistair Cockburn. *Crystal Clear, A Human-Powered Methodology for Small Teams*. Addison-Wesley Professional, October 2004. ISBN 0-201-69947-8.

Edgar Frank Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970. ISSN 0001-0782. doi: 10.1145/362384.362685.

David Cohen, Mikael Lindvall, and Patricia Costa. An Introduction to Agile Methods. *Advances in Computers*, 62:1–66, 2004. doi: 10.1016/S0065-2458(03)62001-2.

Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. MAD Skills: New Analysis Practices for Big Data. *The Proceedings of the VLDB Endowment*, 2(2):1481–1492, 2009.

Douglas Comer. The Difficulty of Optimum Index Selection. *ACM Transactions on Database Systems*, 3(4):440–445, 1978. ISSN 0362-5915. doi: 10.1145/320289.320296.

Douglas Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979. doi: 10.1145/356770.356776.

George P. Copeland and Setrag Khoshafian. A Decomposition Storage Model. In Shamkant B. Navathe, editor, *SIGMOD'85, Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Austin, Texas, May 28-31, 1985*, pages 268–279. ACM Press, 1985. doi: 10.1145/318898.318923.

Douglas Crockford. The application/json Media Type for JavaScript Object Notation (JSON), RFC 4627. `http://tools.ietf.org/html/rfc4627`, July 2006.

Philippe Cudré-Mauroux, Eugene Wu, and Samuel Madden. The Case for RodentStore: An Adaptive, Declarative Storage System. In *CIDR'09, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*. www.cidrdb.org, 2009.

Conor Cunningham, Goetz Graefe, and César A. Galindo-Legaria. PIVOT and UNPIVOT: Optimization and Execution Strategies in an RDBMS. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *VLDB'04, Proceedings of 30th International Conference on Very Large Data Bases, August 31 - September 3, 2004, Toronto, Canada*, pages 998–1009, 2004. ISBN 0-12-088469-0.

Faiz Currim and Sudha Ram. When Entities Are Types: Effectively Modeling Type-Instantiation Relationships. In Juan Trujillo, Gillian Dobbie, Hannu Kangassalo, Sven Hartmann, Markus Kirchberg, Matti Rossi, Iris Reinhartz-Berger, Esteban Zimányi, and Flavius Frasincar, editors, *ERW'10, Advances in Conceptual Modeling - Applications and Challenges, ER 2010 Workshops ACM-L, CMLSA, CMS, DE@ER, FP-UML, SeCoGIS, WISM, Vancouver, BC, Canada, November 1-4, 2010. Proceedings 2010*, volume 6413 of *Lecture Notes in Computer Science*. Springer, 2010. ISBN 978-3-642-16384-5. doi: 10.1007/978-3-642-16385-2_18.

Benoît Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zaït, and Mohamed Ziauddin. Automatic SQL Tuning in Oracle 10g. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *VLDB'04, Proceedings of 30th International Conference on Very Large Data Bases, August 31 - September 3, 2004, Toronto, Canada*, pages 1098–1109. Morgan Kaufmann, 2004. ISBN 0-12-088469-0.

Ole-Johan Dahl, Bjørm Myhrhaug, and Kristen Nygaard. *Common Base Language*. Norwegian Computing Center, 1970.

C. J. Date. A Critique of the SQL Database Language. *SIGMOD Record*, 14(3):8–54, 1984.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *SOSP'07, Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 205–220. ACM, 2007. ISBN 978-1-59593-591-5. doi: 10.1145/1294261.1294281.

Amr El-Helw, Kenneth A. Ross, Bishwaranjan Bhattacharjee, Christian A. Lang, and George A. Mihaila. Column-Oriented Query Processing for Row Stores. In Il-Yeol Song, Alfredo Cuzzocrea, and Karen C. Davis, editors, *DOLAP'11, ACM 14th International Workshop on Data Warehousing and OLAP, Glasgow, United Kingdom, October 28, 2011, Proceedings*, pages 67–74. ACM, 2011. ISBN 978-1-4503-0963-9. doi: 10.1145/2064676.2064689.

Endeca. Endeca Information Access Platform, 2008.

M. T. Fang, Richard C. T. Lee, and Chin-Chen Chang. The Idea of De-Clustering and its Applications. In Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi, editors, *VLDB'86, Proceedings of 12th International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 181–188. Morgan Kaufmann, 1986. ISBN 0-934613-18-4.

Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA Database - Data Management for Modern Business Applications. *SIGMOD Record*, 40(4):45–51, 2011. doi: 10.1145/2094114.2094126.

Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA Database – An Architecture Overview. *IEEE Data(base) Engineering Bulletin*, 35(1):28–33, 2012.

Sheldon J. Finkelstein, Mario Schkolnick, and Paolo Tiberio. Physical Database Design for Relational Databases. *ACM Transactions on Database Systems*, 13(1):91–128, 1988. ISSN 0362-5915. doi: 10.1145/42201.42205.

Franclin S. Foping, Ioannis M. Dokas, John Feehan, and Syed Imran. A New Hybrid Schema-Sharing Technique for Multitenant Applications. In Bill Grosky, Frédéric Andrès, and Pit Pichappan, editors, *ICDIM'09, Fourth IEEE International Conference on Digital Information Management, ICDIM 2009, November 1-4, 2009, University of Michigan, Ann Arbor, Michigan, USA*, pages 211–216. IEEE, 2009. ISBN 978-1-4244-4253-9. doi: 10.1109/ICDIM.2009.5356775.

Michael J. Franklin, Alon Y. Halevy, and David Maier. From Databases to Dataspaces: A New Abstraction for Information Management. *SIGMOD Record*, 34(4):27–33, 2005. doi: 10.1145/1107499.1107502.

Carol Friedman, George Hripcsak, Stephen B. Johnson, James J. Cimino, and Paul D. Clayton. A Generalized Relational Schema for an Integrated Clinical Patient Database. In Randolph A. Miller, editor, *SCAMC'90, Proceedings of the 14th Annual Symposium on Computer Application in Medical Care, November 4-7, 1990, Washington DC*, pages 335–339, 1990.

John Gantz, Christopher Chute, Alex Manfrediz, Stephen Minton, David Reinsel, Wolfgang Schlichting, and Anna Toncheva. The Diverse and Exploding Digital Universe: An Updated Forecast of Worldwide Information Growth Through 2011. IDC White Paper, March 2008.

Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1979. ISBN 0716710455.

James Gleick. *Faster: The Acceleration of Just About Everything.* Pantheon Books, New York, NY, USA, 1999. ISBN 0-679-40837-1.

Goetz Graefe. Dynamic Query Evaluation Plans: Some Course Corrections? *IEEE Data(base) Engineering Bulletin*, 23(2):3–6, 2000.

Goetz Graefe. Sorting And Indexing With Partitioned B-Trees. In *CIDR'03, First Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 5-8, 2003*, 2003.

Goetz Graefe and Harumi A. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In Ioana Manolescu, Stefano Spaccapietra, Jens Teubner, Masaru Kitsuregawa, Alain Léger, Felix Naumann, Anastasia Ailamaki, and Fatma Özcan, editors, *EDBT'10, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, volume 426 of *ACM International Conference Proceeding Series*, pages 371–381. ACM, 2010a. ISBN 978-1-60558-945-9. doi: 10.1145/1739041.1739087.

Goetz Graefe and Harumi A. Kuno. Adaptive indexing for relational keys. In *SMDB'10, 5th International Workshop on Self Managing Database Systems, March 1, 2010, Long Beach, California, USA*, pages 69–74. IEEE, 2010b. doi: 10.1109/ICDEW.2010.5452743.

Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, and Samuel Madden. HYRISE - A Main Memory Hybrid Storage Engine. *The Proceedings of the VLDB Endowment*, 4(2):105–116, 2010.

Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Index Selection for OLAP. In W. A. Gray and Per-Åke Larson, editors, *ICDE'97, Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997, Birmingham, U.K*, pages 208–219. IEEE Computer Society, 1997. ISBN 0-8186-7807-0. doi: 10.1109/ICDE.1997.581755.

Dirk Habich, Matthias Boehm, Maik Thiele, Benjamin Schlegel, Ulrike Fischer, Hannes Voigt, and Wolfgang Lehner. Next Generation Database Programming and Execution Environment. In *DBLP'11, The 13th International Symposium on Database Programming Languages, August 29th, 2011, Seattle, Washington, USA*. VLDB, 2011.

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*, 5(6):502–513, 2012.

Michael Hammer and Arvola Chan. Index Selection in a Self-Adaptive Data Base Management System. In James B. Rothnie, editor, *SIGMOD'76, Proceedings of the 1976 ACM SIGMOD International Conference on Management of Data, Washington, D.C., June 2-4, 1976*, pages 1–8. ACM, 1976. doi: 10.1145/509383.509385.

Richard A. Hankins and Jignesh M. Patel. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. In Johann Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer, editors, *VLDB'03, Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany*, pages 417–428. Morgan Kaufmann, 2003. ISBN 0-12-722442-4.

Michael Hatzopoulos and John G. Kollias. Towards the Optimal Secondary Index Organisation and Secondary Index Selection. *The Computer Journal*, 28(5):524–529, 1985. doi: 10.1093/comjnl/28.5.524.

Kai Herrmann, Hannes Voigt, and Wolfgang Lehner. Online Horizontal Partitioning of Heterogeneous Data. *it - Information Technology*, 56(1):4–12, 2014. doi: 10.1515/itit-2014-1015.

Tony Hey, Stewart Tansley, and Kristin M. Tolle. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009. ISBN 978-0982544204.

Martin Hilbert and Priscila López. The World's Technological Capacity to Store, Communicate, and Compute Information. *Science*, 332(6025):60–65, April 2011. doi: 10.1126/science.1200970.

Jeffrey A. Hoffer and Dennis G. Severance. The Use of Cluster Analysis in Physical Data Base Design. In Douglas S. Kerr, editor, *VLDB'75, Proceedings of the International Conference on Very Large Data Bases, September 22-24, 1975, Framingham, Massachusetts, USA*, pages 69–86. ACM, 1975.

Michael Hunger. Cool first Neo4j 2.0 milestone – Now with Labels and "real" Indexes. Better Software Development Blog, `http://jexp.de/blog/2013/04/cool-first-neo4j-2-0-milestone-now-with-labels-and-real-indexes/`, April 2013.

IBM. Autonomic Computing Concepts. IBM White Paper, 2001.

Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database Cracking. In *CIDR'07, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pages 68–78, 2007.

Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Self-organizing Tuple Reconstruction in Column-stores. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *SIGMOD'09, Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29-July 2, 2009*, pages 297–308, 2009. ISBN 978-1-60558-551-2. doi: 10.1145/1559845.1559878.

Stratos Idreos, Stefan Manegold, Harumi A. Kuno, and Goetz Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *The Proceedings of the VLDB Endowment*, 4(9):585–597, 2011.

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*, 9(2):135–143, 1983. doi: 10.1109/TSE.1983.236458.

ISO/IEC. ISO/IEC 9075-x:200x (Information technology – Database languages – SQL), 2006.

Dean Jacobs. Enterprise Software as Service. *ACM Queue*, 3(6):36–42, 2005. doi: 10.1145/1080862.1080875.

Dean Jacobs and Stefan Aulbach. Ruminations on Multi-Tenant Databases. In Alfons Kemper, Harald Schöning, Thomas Rose, Matthias Jarke, Thomas Seidl, Christoph Quix, and Christoph Brochhaus, editors, *BTW'07, Datenbanksysteme in Business, Technologie und Web (BTW 2007), 12. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), Proceedings, 7.-9. März 2007, Aachen, Germany*, volume 103 of *LNI*, pages 514–521. GI, 2007. ISBN 978-3-88579-197-3.

Gerhard Jaeschke and Hans-Jörg Schek. Remarks on the Algebra of Non First Normal Form Relations. In *PODS'82, Proceedings of the ACM Symposium on Principles of Database Systems, March 29-31, 1982, Los Angeles, California*, pages 124–138. ACM, 1982. doi: 10.1145/588111.588133.

Alekh Jindal and Jens Dittrich. Relax and Let the Database Do the Partitioning Online. In Malú Castellanos, Umeshwar Dayal, and Wolfgang Lehner, editors, *BIRTE'11, Enabling Real-Time Business Intelligence - 5th International Workshop, BIRTE 2011, Held at the 37th International Conference on Very Large Databases, VLDB 2011, Seattle, WA, USA, September 2, 2011, Revised Selected Papers*, volume 126 of *Lecture Notes in Business Information Processing*, pages 65–80. Springer, 2011. ISBN 978-3-642-33499-3. doi: 10.1007/978-3-642-33500-6_5.

Alekh Jindal, Felix Schuhknecht, Jens Dittrich, Karen Khachatryan, and Alexander Bunte. How Achaeans Would Construct Columns in Troy. In *CIDR'13, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*, 2013.

Bradford C. Johnson, James M. Manyika, and Lareina A. Yee. The next revolution in interactions. *The McKinsey Quarterly*, (4):21–33, 2005.

George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel Hypergraph Partitioning: Application in VLSI Domain. In *DAC'97, Proceedings of the 34th annual Design Automation Conference*, pages 526–529, 1997. doi: 10.1145/266021.266273.

George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel Hypergraph Partitioning: Applications in VLSI Domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1):69–79, 1999. doi: 10.1109/92.748202.

Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *ICDE'11, Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 195–206. IEEE Computer Society, 2011. ISBN 978-1-4244-8958-9. doi: 10.1109/ICDE.2011.5767867.

Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *IEEE Computer Magazine*, 36(1):41–50, 2003. doi: 10.1109/MC.2003.1160055.

David Kernert, Frank Köhler, and Wolfgang Lehner. Bringing Linear Algebra Objects to Life in a Column-Oriented In-Memory Database. In *IMDM'13, 1st Workshop on In-memory Data Management and Analytics*, 2013.

Martin L. Kersten and Stefan Manegold. Cracking the Database Store. In *CIDR'05, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pages 213–224, 2005.

Gaye Kiely and Brian Fitzgerald. An Investigation of the Use of Methods within Information Systems Development Projects. *The Electronic Journal of Information Systems in Developing Countries*, 22(4):1–13, 2005. ISSN 1681-4835.

Thomas Kissinger, Hannes Voigt, and Wolfgang Lehner. SMIX Live – A Self-Managing Index Infrastructure for Dynamic Workloads. In Anastasios Kementsietsidis and Marcos Antonio Vaz Salles, editors, *ICDE'12, IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 1225–1228. IEEE Computer Society, 2012. ISBN 978-0-7685-4747-3. doi: 10.1109/ICDE.2012.9.

Ray Kurzweil. The Law of Accelerating Returns. `http://www.kurzweilai.net/the-law-of-accelerating-returns`, March 2001.

Avinash Lakshman and Prashant Malik. Cassandra - A Decentralized Structured Storage System. *Operating Systems Review*, 44(2):35–40, 2010. doi: 10.1145/1773912.1773922.

Per-Åke Larson, Cipri Clinciu, Eric N. Hanson, Artem Oks, Susan L. Price, Srikumar Rangarajan, Aleksandras Surna, and Qingqing Zhou. SQL Server Column Store Indexes. In Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis, editors, *SIGMOD'11, Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 1177–1184. ACM, 2011. ISBN 978-1-4503-0661-4. doi: 10.1145/1989323.1989448.

Pierre-Alain Laur, Florent Masseglia, and Pascal Poncelet. Schema mining: Finding structural regularity among semistructured data. In Djamel A. Zighed, Henryk Jan Komorowski, and Jan M. Zytkow, editors, *PKDD'00, Principles of Data Mining and Knowledge Discovery, 4th European Conference, PKDD 2000, Lyon, France, September 13-16, 2000, Proceedings*, volume 1910 of *Lecture Notes in Computer Science*, pages 498–503. Springer, 2000. ISBN 3-540-41066-X. doi: 10.1007/3-540-45372-5_57.

Neal Leavitt. Will NoSQL Databases Live Up to Their Promise? *IEEE Computer Magazine*, 43(2): 12–14, 2010. doi: 10.1109/MC.2010.58.

Thomas Legler, Wolfgang Lehner, and Andrew Ross. Data Mining with the SAP Netweaver BI Accelerator. In Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors, *VLDB'06, Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 1059–1068. ACM, 2006. ISBN 1-59593-385-9.

Wolfgang Lehner. *Datenbanktechnologie für Data-Warehouse-Systeme: Konzepte und Methoden.* dpunkt-Verlag Heidelberg, September 2002. ISBN 3-8986-4177-5.

Wolfgang Lehner and Kai-Uwe Sattler. *Web-Scale Data Management for the Cloud*, chapter 2.4: Schema Virtualization. Springer, March 2013. ISBN 978-1-4614-6855-4.

Tom Leighton, Fillia Makedon, and Spyros Tragoudas. Approximation Algorithms for VLSI Partition Problems. In *ISCAS'90, IEEE International Symposium on Circuits and Systems, New Orleans, LA, 1-3 May, 1990*, 1990. doi: 10.1109/ISCAS.1990.112608.

Christian Lemke, Kai-Uwe Sattler, Franz Faerber, and Alexander Zeier. Speeding Up Queries in Column Stores – A Case for Compression. In *DaWak'10, Data Warehousing and Knowledge Discovery, 12th International Conference, DAWAK 2010, Bilbao, Spain, August/September 2010.*, volume 6263 of *Lecture Notes in Computer Science*, pages 117–129. Springer, 2010. ISBN 978-3-642-15104-0. doi: 10.1007/978-3-642-15105-7_10.

Sam Lightstone and Bishwaranjan Bhattacharjee. Automating the design of multi-dimensional clustering tables in relational databases. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *VLDB'04, Proceedings of 30th International Conference on Very Large Data Bases, August 31 - September 3, 2004, Toronto, Canada*, pages 1170–1181. Morgan Kaufmann, 2004. ISBN 0-12-088469-0.

Duen-Ren Liu and Shashi Shekhar. Partitioning Similarity Graphs: A Framework for Declustering Problems. *Information Systems*, 21(6):475–496, 1996. doi: 10.1016/0306-4379(96)00024-5.

Ling Liu and M. Tamer Özsu. *Encyclopedia of Database Systems.* Springer US, 2009. ISBN 978-0-387-35544-3, 978-0-387-39940-9.

Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *The VLDB Journal – The International Journal on Very Large Data Bases*, 9(3):231–246, 2000. doi: 10.1007/s007780000031.

Salvatore T. March and Dennis G. Severance. The Determination of Efficient Record Segmentations and Blocking Factors for Shared Data Files. *ACM Transactions on Database Systems*, 2(3):279–296, 1977. doi: 10.1145/320557.320574.

John McCarthy, R. Brayton, Daniel J. Edwards, P. Fox amd L. Hodes, D. Luckham, K. Maling, D. Park, and S. Russell. *LISP I Programmers Manual.* M.I.T. Computation Center and Research Laboratory of Electronics, Boston, Massachusetts, March 1960.

John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual.* MIT Press, 1962. ISBN 0-262-13011-4.

Joseph McKendrick. Big Data, Big Challenges, Big Opportunities: 2012 IOUG Big Data Strategies Survey. IOUG, September 2012.

Joseph McKendrick. Big Data Visionaries: 2013 Data Science Skills Survey. IOUG, February 2013.

C. Mohan. History Repeats Itself: Sensible and NonsenSQL Aspects of the NoSQL Hoopla. In Giovanna Guerrini and Norman W. Paton, editors, *EDBT'13, Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18–22, 2013*, pages 11–16. ACM, 2013. ISBN 978-1-4503-1597-5. doi: 10.1145/2452376.2452378.

C. Mohan, Donald J. Haderle, Yun Wang, and Josephine M. Cheng. Single Table Access Using Multiple Indexes: Optimization, Execution, and Concurrency Control Techniques. In François Bancilhon, Costantino Thanos, and Dennis Tsichritzis, editors, *EDBT'90, International Conference on Extending Database Technology, Venice, Italy, March 26-30, 1990, Proceedings*, volume 416 of *Lecture Notes in Computer Science*, pages 29–43. Springer, 1990. ISBN 3-540-52291-3. doi: 10.1007/BFb0022162.

Curt Monash. Terminology: Investigative analytics. DBMS2 Blog: `http://www.dbms2.com/2011/03/03/investigative-analytics/`, March 2011.

Curt Monash. Data model churn. DBMS2 Blog: `http://www.dbms2.com/2013/08/04/data-model-churn/`, August 2013a.

Curt Monash. What matters in investigative analytics? DBMS2 Blog: `http://www.dbms2.com/2013/10/06/what-matters-in-investigative-analytics/`, October 2013b.

David A. Moon. Object-Oriented Programming with Flavors. In Norman K. Meyrowitz, editor, *OOPSLA'86, Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86), Portland, Oregon, Proceedings*, pages 1–8. ACM, 1986. ISBN 0-89791-204-7. doi: 10.1145/28697.28698.

Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

Thomas Mueller. H2 Database. `http://www.h2database.com`, 2012.

Sundara Nagarajan. Guest Editor's Introduction: Data Storage Evolution. *Computing Now, Special Issue*, March 2011.

Neo Technology. *The Neo4j Manual v2.0.0-M02*, chapter 3.4: Labels. Neo Technology, 2013a.

Neo Technology. Neo4j. `http://neo4j.org/`, 2013b.

Svetlozar Nestorov, Serge Abiteboul, and Rajeev Motwani. Extracting Schema from Semistructured Data. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD'98, Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, Seattle, Washington, USA, June 2-4, 1998*, pages 295–306. ACM Press, 1998. ISBN 0-89791-995-5. doi: 10.1145/276304.276331.

Matthias Nicola and Bert Van der Linden. Native XML Support in DB2 Universal Database. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *VLDB'05, Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 1164–1174. ACM, 2005. ISBN 1-59593-154-6, 1-59593-177-5.

Taiichi Ohno. *The Toyota Production System: Beyond Large-Scale Production.* Productivity Press, 1988. ISBN 0-915299-14-3.

OMG. Unified Modeling Language™ (OMG UML), Infrastructure. `http://www.omg.org/cgi-bin/doc?formal/2009-02-04`, February 2009. formal/2009-02-04.

Beng Chin Ooi, Bei Yu, and Guoliang Li. One Table Stores All: Enabling Painless FreeandEasy Data Publishing and Sharing. In *CIDR'07, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pages 142–153, 2007.

Stratos Papadomanolakis, Debabrata Dash, and Anastassia Ailamaki. Efficient Use of the Query Optimizer for Automated Database Design. In Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *VLDB'07, Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 1093–1104. ACM, 2007. ISBN 978-1-59593-649-3.

Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object Exchange Across Heterogeneous Information Sources. In Philip S. Yu and Arbee L. P. Chen, editors, *ICDE'95, Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, pages 251–260. IEEE Computer Society, 1995. ISBN 0-8186-6910-1. doi: 10.1109/ICDE.1995.380386.

Daniel H. Pink. *A Whole New Mind: Why Right-Brainers Will Rule the Future*. Riverhead Books, 2006.

Daniel H. Pink. *Drive*. Riverhead Books, 2009.

Peter Pistor and F. Andersen. Designing A Generalized NF2 Model with an SQL-Type Language Interface. In Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi, editors, *VLDB'86, Proceedings of 12th International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 278–285. Morgan Kaufmann, 1986. ISBN 0-934613-18-4.

Hasso Plattner. A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *SIGMOD'09, Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 1–2. ACM, 2009. ISBN 978-1-60558-551-2. doi: 10.1145/1559845.1559846.

PostgreSQL Global Development Group. *PostgreSQL 9.2.4 Documentation*, chapter 56.6: Database Page Layout. PostgreSQL Global Development Group, 2013.

Walter W. Powell and Kaisa Snellman. The Knowledge Economy. *Annual Review of Sociology*, 30: 199–220, 2004. doi: 10.1146/annurev.soc.29.010202.100037.

Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. A Case for Fractured Mirrors. In *VLDB'02, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 430–441. Morgan Kaufmann, 2002.

Vijayshankar Raman, Lin Qiao, Wei Han, Inderpal Narang, Ying-Lin Chen, Kou-Horng Yang, and Fen-Ling Ling. Lazy, Adaptive RID-List Intersection, and Its Application to Index Anding. In *SIGMOD'07, Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 773–784, 2007. doi: 10.1145/1247480.1247566.

Jun Rao and Kenneth A. Ross. Making B^+-Trees Cache Conscious in Main Memory. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *SIGMOD'00, Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 475–486. ACM, 2000. ISBN 1-58113-218-2. doi: 10.1145/342009.335449.

Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy M. Lohman. Automating physical database design in a parallel database. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *SIGMOD'02, Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pages 558–569. ACM, 2002. ISBN 1-58113-497-5. doi: 10.1145/564691.564757.

Philip Rathle. Nodes are people, too. Neo4j Blog, `http://blog.neo4j.org/2013/04/nodes-are-people-too.html`, April 2013.

Marko A. Rodriguez and Peter Neubauer. Constructions from Dots and Lines. *The Computing Research Repository*, abs/1006.2361(6), June 2010. doi: http://arxiv.org/abs/1006.2361.

Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. The graph story of the sap hana database. In Volker Markl, Gunter Saake, Kai-Uwe Sattler, Gregor Hackenbroich, Bernhard Mitschang, Theo Härder, and Veit Köppen, editors, *BTW'13, Datenbanksysteme für Business, Technologie und Web (BTW), 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 11.-15.3.2013 in Magdeburg, Germany. Proceedings*, volume 214 of *LNI*, pages 403–420. GI, 2013. ISBN 978-3-88579-608-4.

Anish Das Sarma, Xin Dong, and Alon Y. Halevy. Bootstrapping Pay-As-You-Go Data Integration Systems. In Jason Tsong-Li Wang, editor, *SIGMOD'08, Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 861–874. ACM, 2008. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376702.

Kai-Uwe Sattler, Ingolf Geist, and Eike Schallehn. QUIET: Continuous Query-driven Index Tuning. In Johann Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer, editors, *VLDB'03, Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany*, pages 1129–1132. Morgan Kaufmann, 2003. ISBN 0-12-722442-4.

Kai-Uwe Sattler, Eike Schallehn, and Ingolf Geist. Autonomous Query-Driven Index Tuning. In *IDEAS'04, Procesdings of the 8th International Database Engineering and Applications Symposium, 7-9 July 2004, Coimbra, Portugal*, pages 439–448. IEEE Computer Society, 2004. ISBN 0-7695-2168-1. doi: 10.1109/IDEAS.2004.15.

Kai-Uwe Sattler, Martin Luehring, Karsten Schmidt, and Eike Schallehn. Autonomous Management of Soft Indexes. In *SMDB'07, 2nd International Workshop on Self-Managing Database Systems, 16 April 2007, Istanbul, Turkey*, 2007. doi: 10.1109/ICDEW.2007.4401028.

Jan Schaffner, Anja Bog, Jens Krüger, and Alexander Zeier. A Hybrid Row-Column OLTP Database Architecture for Operational Reporting. In *BIRTE'08, Informal Proceedings of the Second International Workshop on Business Intelligence for the Real-Time Enterprise, BIRTE 2008, in conjunction with VLDB'08, August 24, 2008, Auckland, New Zealand*, 2008.

Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable Units of Behaviour. In Luca Cardelli, editor, *ECOOP'03 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274. Springer, 2003. ISBN 3-540-40531-3. doi: 10.1007/978-3-540-45070-2_12.

Hans-Jörg Schek and Peter Pistor. Data Structures for an Integrated Data Base Management and Information Retrieval System. In *VLDB'82, Proceedings of Eigth International Conference on Very Large Data Bases, September 8-10, 1982, Mexico City, Mexico, Proceedings*, pages 197–207. Morgan Kaufmann, 1982. ISBN 0-934613-14-1.

Mario Schkolnick. A Survey of Physical Database Design Methodology and Techniques. In S. Bing Yao, editor, *VLDB'78, Proceedings of Fourth International Conference on Very Large Data Bases, September 13-15, 1978, West Berlin, Germany*, pages 474–487. IEEE Computer Society, 1978.

Albrecht Schmidt, Martin L. Kersten, Menzo Windhouwer, and Florian Waas. Efficient Relational Storage and Retrieval of XML Documents. In Dan Suciu and Gottfried Vossen, editors, *WebDB'00, The World Wide Web and Databases, Third International Workshop WebDB 2000, Dallas, Texas, USA, May 18-19, 2000, Selected Papers*, volume 1997 of *Lecture Notes in Computer Science*, pages 137–150. Springer, 2001. ISBN 3-540-41826-1. doi: 10.1007/3-540-45271-0_9.

Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. COLT: Continuous On-Line Database Tuning. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *SIGMOD'06, Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 793–795. ACM, 2006. ISBN 1-59593-256-9. doi: 10.1145/1142473.1142592.

Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. On-Line Index Selection for Shifting Workloads. In *SMDB'07, 2nd International Workshop on Self-Managing Database Systems, 16 April 2007, Istanbul, Turkey*, 2007. doi: 10.1109/ICDEW.2007.4401029.

Ernst J. Schuegraf. Compression of Large Inverted Files with Hyperbolic Term Distribution. *Information Processing & Management*, 12(6):377–384, 1976. doi: 10.1016/0306-4573(76)90035-2.

Ken Schwaber. SCRUM Development Process. In Jeff Sutherland, Cory Casanave, Joaquin Miller, Philip Patel, and Glenn Hollowell, editors, *Business Object Design and Implementation, OOPSLA'95 Workshop Proceedings 16 October 1995, Austin, Texas*, pages 117–134. Springer London, 1997. ISBN 978-3-540-76096-2. doi: 10.1007/978-1-4471-0947-1_11.

Praveen Seshadri and Arun N. Swami. Generalized Partial Indexes. In Philip S. Yu and Arbee L. P. Chen, editors, *ICDE'95, Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, pages 420–427. IEEE Computer Society, 1995. ISBN 0-8186-6910-1. doi: 10.1109/ICDE.1995.380355.

Nigel Shadbolt, Tim Berners-Lee, and Wendy Hall. The Semantic Web Revisited. *IEEE Intelligent Systems*, 21(3):96–101, 2006. ISSN 1541-1672. doi: 10.1109/MIS.2006.62.

Yakov Shafranovich. Common Format and MIME Type for Comma-Separated Values (CSV) Files, RFC 4180. `http://tools.ietf.org/html/rfc4180`, October 2005.

Friedrich Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data & Knowledge Engineering*, 35(1):83–106, 2000. doi: 10.1016/S0169-023X(00)00023-9.

David Stodder. TDWI Best Practices Report: Customer Analytics in the Age of Social Media. TDWI Research, July 2012.

Michael Stonebraker. The Case for Partial Indexes. *SIGMOD Record*, 18(4):4–11, 1989. doi: 10.1145/74120.74121.

Michael Stonebraker. Stonebraker on NoSQL and enterprises. *Communications of the ACM*, 54(8): 10–11, 2011. doi: 10.1145/1978542.1978546.

Hirotaka Takeuchi and Ikujiro Nonaka. The New New Product Development Game. *Harvard Business Review*, 64(1):137–146, 1986.

Nassim Nicholas Taleb. *The Black Swan*, chapter 11: How to Look For Bird Poop, pages 165–189. Random House, 2010.

Toby J. Teorey, Dongqing Yang, and James P. Fry. A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model. *ACM Computing Surveys*, 18(2): 197–222, 1986. doi: 10.1145/7474.7475.

Transaction Processing Performance Council. TPC Benchmark™ H, Revision 2.16.0. `http://www.tpc.org/tpch/`, June 2013.

Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis. The GMAP: A Versatile Tool for Physical Data Independence. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 367–378. Morgan Kaufmann, 1994. ISBN 1-55860-153-8.

Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis. The GMAP: A Versatile Tool for Physical Data Independence. *The VLDB Journal – The International Journal on Very Large Data Bases*, 5(2):101–118, 1996. doi: 10.1007/s007780050018.

Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *ICDE'00, Proceedings of the 16th International Conference on Data Engineering, 28 February - 3 March, 2000, San Diego, California, USA*, pages 101–110. IEEE Computer Society, 2000. doi: 10.1109/ICDE.2000.839397.

Yannis Vassiliou. Null Values in Data Base Management: A Denotational Semantics Approach. In Philip A. Bernstein, editor, *SIGMOD'79, Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 30 - June 1*, pages 162–169. ACM, 1979. ISBN 0-89791-001-X. doi: 10.1145/582095.582123.

Inc VersionOne. 7th Annual State of Agile Development Survey, 2012.

Fernanda B. Viégas and Martin Wattenberg. Artistic Data Visualization: Beyond Visual Analytics. In Douglas Schuler, editor, *OCSC'07, Online Communities and Social Computing, Second International Conference, OCSC 2007, Held as Part of HCI International 2007, Beijing, China, July 22-27, 2007, Proceedings*, volume 4564 of *Lecture Notes in Computer Science*, pages 182–191. Springer, 2007. ISBN 978-3-540-73256-3. doi: 10.1007/978-3-540-73257-0_21.

Hannes Voigt, Wolfgang Lehner, and Kenneth Salem. Constrained Dynamic Physical Database Design. In *SMDB'08, 3rd International Workshop on Self Managing Database Systems, 7 April 2008, Cancun, Mexico*. IEEE Computer Society, 2008. doi: 10.1109/ICDEW.2008.4498286.

Hannes Voigt, Tobias Jäkel, Thomas Kissinger, and Wolfgang Lehner. Adaptive Index Buffer. In *SMDB'12, 7th International Workshop on Self Managing Database Systems, 1 April 2012, Washington, DC, USA*, pages 308–314. IEEE Computer Society, 2012. doi: 10.1109/ICDEW.2012.39.

Hannes Voigt, Thomas Kissinger, and Wolfgang Lehner. SMIX – Self-Managing Indexes for Dynamic Workloads. In Alex Szalay, Tamas Budavari, Magdalena Balazinska, Alexandra Meliou, and Ahmet Sacan, editors, *SSDBM'13, Conference on Scientific and Statistical Database Management, Baltimore, MD, USA, July 29 - 31, 2013*, pages 24–35. ACM, 2013. ISBN 978-1-4503-1921-8. doi: 10.1145/2484838.2484862.

W3C. Resource Description Framework (RDF): Concepts and Abstract Syntax. `http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/`, February 2004a.

W3C. RDF Vocabulary Description Language 1.0: RDF Schema. `http://www.w3.org/TR/2004/REC-rdf-schema-20040210/`, February 2004b.

W3C. Extensible Markup Language (XML) 1.0 (Fifth Edition). `http://www.w3.org/TR/2008/REC-xml-20081126/`, November 2008.

W3C. XQuery 1.0: An XML Query Language (Second Edition). `http://www.w3.org/TR/2010/REC-xquery-20101214/`, December 2010a.

W3C. XML Linking Language (XLink) Version 1.1. `http://www.w3.org/TR/2010/REC-xlink11-20100506/`, May 2010b.

W3C. XML Schema Definition Language (XSD) 1.1 Part 1: Structures. `http://www.w3.org/TR/2011/CR-xmlschema11-1-20110721/`, July 2011a.

W3C. XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes. `http://www.w3.org/TR/2011/CR-xmlschema11-2-20110721/`, July 2011b.

Ke Wang and Huiqing Liu. Discovering Typical Structures of Documents: A Road Map Approach. In *SIGIR'98: Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, August 24-28 1998, Melbourne, Australia*, pages 146–154. ACM, 1998. doi: 10.1145/290941.290982.

Ke Wang and Huiqing Liu. Discovering Structural Association of Semistructured Data. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):353–371, 2000. doi: 10.1109/69.846290.

David Graham Wastell. The fetish of technique: methodology as a social defence. *Information Systems Journal*, 6(1):25–40, 1996. doi: 10.1046/j.1365-2575.1996.00104.x.

Gerhard Weikum, Christof Hasse, Alex Moenkeberg, and Peter Zabback. The COMFORT Automatic Tuning Project, Invited Project Review. *Information Systems*, 19(5):381–432, 1994. doi: 10.1016/0306-4379(94)90004-3.

Craig D. Weissman and Steve Bobrowski. The Design of the Force.com Multitenant Internet Application Development Platform. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *SIGMOD'09, Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 889–896. ACM, 2009. ISBN 978-1-60558-551-2. doi: 10.1145/1559845.1559942.

Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. The Implementation and Performance of Compressed Databases. *SIGMOD Record*, 29(3):55–67, 2000. doi: 10.1145/362084.362137.

Marianne Winslett. Bruce Lindsay Speaks Out on System R, Benchmarking, Life as an IBM Fellow, the Power of DBAs in the Old Days, Why Performance Still Matters, Heisenbugs, Why He Still Writes Code, Singing Pigs, and More. *SIGMOD Record*, 34(2):71–79, 2005. doi: 10.1145/1083784.1083803.

Eugene Wu and Samuel Madden. Partitioning Techniques for Fine-grained Indexing. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *ICDE'11, Proceedings of the 27th International Conference on Data Engineering, April 11-16, 2011, Hannover, Germany*, pages 1127–1138. IEEE Computer Society, 2011. ISBN 978-1-4244-8958-9. doi: 10.1109/ICDE.2011.5767830.

Rui Zhang, Richard T. Snodgrass, and Saumya Debray. Micro-Specialization in DBMSes. In Anastasios Kementsietsidis and Marcos Antonio Vaz Salles, editors, *ICDE'12, IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 690–701. IEEE Computer Society, 2012a. ISBN 978-0-7695-4747-3. doi: 10.1109/ICDE.2012.110.

Rui Zhang, Richard T. Snodgrass, and Saumya Debray. Application of Micro-specialization to Query Evaluation Operators. In *SMDB'12, 7th International Workshop on Self Managing Database Systems, 1 April 2012, Washington, DC, USA*, pages 315–321. IEEE Computer Society, 2012b. ISBN 978-1-4673-1640-8. doi: 10.1109/ICDEW.2012.43.

Jingren Zhou, Per-Åke Larson, and Hicham G. Elmongui. Lazy Maintenance of Materialized Views. In Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *VLDB'07, Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 231–242. ACM, 2007. ISBN 978-1-59593-649-3.

Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and

K. Bernhard Schiefer, editors, *VLDB'04, Proceedings of 30th International Conference on Very Large Data Bases, August 31 - September 3, 2004, Toronto, Canada*, pages 1087–1097. Morgan Kaufmann, 2004. ISBN 0-12-088469-0.

List of Figures

List of Tables