Querying Sparse Matrices for Information Retrieval

Roberto Cornacchia

Querying Sparse Matrices for Information Retrieval

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen
op dinsdag 29 mei 2012 om 12.30 uur

door Roberto CORNACCHIA
Dottore Magistrale, Università di Bologna, Italië
geboren te Rimini, Italië

Dit proefschrift is goedgekeurd door de promotor:

Prof. dr. ir. A.P. de Vries

The research reported in this thesis has been carried out partially at CWI, the Dutch national research centre for mathematics and computer science, within research groups *Database Architectures* and *Interactive Information Access* in the cluster *Information Systems*.

The research reported in this thesis has been carried out partially at Spinque B.V., a CWI spin-off company.

The research reported in this thesis has been funded by the MultimediaN N5 project.

SIKS Dissertation Series No. 2012-21
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

Cover design: Roland MacDonald (www.rolandtheillustrator.com)

ISBN: 978-90-6464-560-0

To my parents, who taught me their best lessons by simply being themselves.

Ai miei genitori, che mi hanno trasmesso i loro migliori insegnamenti semplicemente essendo se stessi.

Contents

Acknowledgements

When I wrote my first email to Arjen de Vries to apply for a PhD position at CWI I had no idea it would have then been so difficult to get rid of him. Luckily, he gave me no reason to regret this and instead many reasons to thank him today. I owe him my heartfelt gratitude for the faith and confidence he always showed in my abilities, without which I would not have succeeded in this endeavour. I am likewise grateful for his genuine and contagious passion for science and intellectual exchange, which is, I believe, one of the reasons for his excellence in research: he values others' opinion to the point that he always hopes to learn something from his own students. I also thank him for the freedom I had to pursue the wrong direction several times. I learnt many important lessons from the mistakes I was allowed to make (during my first year, he knowingly let me get trapped for days in one of the most silly ideas I ever had). Finally, I should point out again that Arjen is not easy to get rid of. I enjoyed the many occasions we had to rest from work together, in particular the nights out with friends and the great sailing weekends.

I wish to express my gratitude to the great scientists who accepted to be members of my promotion committee: Prof.dr. Peter Apers, Prof.dr. Norbert Fuhr, Prof.dr. Martin Kersten, Prof.dr.ir. Geert-Jan Houben and Prof.dr.ir. Marcel Reinders. I do hope you found my work interesting.

During the years I spent working at CWI I had the opportunity to meet and work with truly talented and inspiring scientists. I owe special thanks to some of them for their active contribution to my work. Alex van Ballegooij and I have shared a considerable part of our research interests and I enjoyed the many brainstorming sessions with him. My research literally builds on his efforts. Marcin Żukowski has always been my officemate, the official reasons being more than one: to be able to play darts every other line of code written, to honour the fact that I have been the very first (happy) user of his superb

creation, MonetDB/X100, and because we soon became good friends. The real reason, however, is that I secretly hoped I would absorb a bit of is being brilliant without any apparent effort. I obviously did not succeed, but it was worth a try. MonetDB and MonetDB/X100 have been fundamental in my research and I want to address special thanks to all the great minds who have contributed to these projects. Martin Kersten had been rightfully described to me as an "excellent scientist and person" before I first met him. Countless researchers have been inspired by his unique way to play with ideas and turn them around, typical of the great dreamers. Peter Boncz must have fallen into a cauldron of magic potion when he was a baby, like Obelix. But then, a superhuman-talent potion. His direct and indirect contributions to my work are invaluable. I especially thank Stefan Manegold for his continuous examples of what good quality work means, and for his patience in answering the many questions I still ask him (I admit, I triggered some of the discussions we had mainly for the pleasure of seeing problems analysed in such a great detail). The same gratitude goes to Niels Nes, Sjoerd Mullender and Fabian Groffen. The excellent work of Sandor Héman has also been crucial for my own results. I have been long enough in that research group to be excused if I cannot mention everyone by name and simply state my thanks to all the other past and present members. The very first years deserve, however, some more words. Exciting and thoughtless times, when partying was still part of the PhD experience. I had some really great time with Georgina, Thijs, Marcin, Arjen, Elena and Dorina, besides many stimulating discussions. I want to address my sincere gratitude to Paolo Ciaccia, who made me walk my first steps in the world of databases during my University years in Italy, encouraged my interest for research and supported me fully when I most needed his help. Wouter, Arjen and Roland have been so kind to help me promptly with details of this thesis that go well beyond my area of expertise (to put it mildly): translations into Dutch and cover design.

My PhD journey has been longer than expected. The fact that this does not particularly bother me is not to be seen as a lack of interest, but rather as the appreciation of the many other great things that happened to me. I would like to thank explicitly all those who did not contribute to my thesis, and especially those who, knowingly or not, have actually slowed me down along the way, because they all added something to my life. During the past few years, I had the pleasure to get to know Wouter Alink better and start with him and Arjen some challenging and exciting professional experiences. Having the opportunity to keep working on interesting projects with them is priceless – and fun! Luckily, I realised very soon the importance of a computer-free part of my existence. A large portion of this is devoted to my passion for dance. A huge, general thank goes to all

those who kept me on a dance floor rather than behind a screen. In particular, to all my teachers, to my friends at the Cotton Club and to my fellow Smokey Feet crew members, Quinten, Sarah, Praveen and Dolinde – thank you guys! To Dorine and Sarah for all the dances and dinners in the early days and for being virtually present when I wrote the first words of this thesis. To Marijn, Laura, Lisa, Ursula, Jojanneke and Dolinde, for all I learnt from you every time we tried to share our passion with others, and for being friends with me.

I had the privilege to meet and get to know Renè and Marja during these years, easily the most sociable and friendly people in Amsterdam.

Some of my best friends are far from here, but I know I can always count on them: Marco, Patty, Piero, Francesca, Alice, Luca. They have never really been left behind. My parents, Renato and Otella, and my siblings, Ernesto, Marco and Daniela, have always been my most solid point of reference and the best examples to refer to. Thanks for your your endless support. Thanks to Patricia and Claudio, for making me feel part of their family and for all the great help they gave us. Two young children are accountable, at the same time, for the largest amount of time stolen to my work and PhD and for the most overwhelming joy I ever felt in my life. Maria and Romeo are undoubtedly the best gift I could ever receive. It is a continuous surprise how much I can learn from them.

There is one notable omission in my acknowledgements so far. I did not mention my wife, Elisa, because she is so special that she wouldn't fit anywhere else. She helped me actively in my research with many useful tips, being an excellent researcher herself. She helped me by taking too many times my home duties on her shoulders to give me more time to work. She helped me with her love when I was frustrated and tired. She also slowed down my research at times, and I'm glad she did. She danced with me and encouraged me to keep doing what I love. She is one of my best friends and she is my family. She loves me, even when I don't deserve it. Thank you Eli, I love you.

Finally, please allow me to thank myself. Because it has not been easy, but I did not give up.

Roberto Cornacchia
Utrecht, 22 April 2012

One often meets his destiny on the road he takes to avoid it.

- Jean de La Fontaine, poet

1 Introduction

In the years in which this thesis is being written and published, computer-assisted search has consolidated as an everyone's daily activity. The amount of information available on the World Wide Web is so large that we no longer wonder *whether* what we need is on the Web, but rather *where* to find it exactly. This is one of the reasons why *search systems* have become so crucial in our life. Allowing ourselves to forget and search repeatedly what we need to find is often more convenient than maintaining a personal list of interesting resources. For the vast majority of non-specialists, the concept of computer-assisted search corresponds to 'to google' (introduced in the Oxford English Dictionary in 2006 [Oxf]), which refers to using the Google search engine [Goo] (or other popular alternatives) to obtain information on the Web.

In spite of what many may think, improving the way we index, search and present information is far from being a dead-end research topic. "Isn't there Google for this already?" is the typical question that a researcher in the field of *information retrieval* is asked by friends and relatives. Luckily, search will evolve much beyond what Google and others can offer today. The author's children will probably smile trying to picture how difficult their dad's life was, when he had to type keywords on the Web in order to find what he needed.

Information retrieval, in its most generic definition, is the science that studies how to find information that is relevant to our needs. Simple real-life scenarios include at least the following entities: a *user* who has an information need (often not precisely defined); a *query* that describes such an information need (e.g. a list of keywords); a *data* repository of diverse data kinds (e.g. text, xml, multimedia); a *retrieval method* that is used to assess the degree of relevance of each piece of information to the user query; a *search system* (e.g. Google), which finally accepts queries from users and finds relevant information in the available data, applying some retrieval methods.

Information retrieval queries represent information needs that in natural language would be expressed as e.g. “news *about* the financial crisis”, “other photos *like* the selected one”, or “the books read by users whose taste is *similar to* mine”. The ability of automated systems to answer such vague queries is probably what attracts us the most, as it matches the lack of completeness and structure that we experience in our life. We usually do not know in the exact detail what we are looking for, and we know certain questions have no exact answers. Typically, search systems use a “best effort” approach to produce a *ranked* list of result items, so that the probability to be pointed to right direction is maximised when looking at the items that are highest in this rank. Unfortunately, instructing an automated system to perform such a “guess” is considerably more difficult than demanding exact answers, as it requires the formalisation and consequent simulation of an inductive process.

Retrieval methods are the formalisation of such inductive processes. For example, observing that a news article contains many repetitions of words ‘swine’ and ‘flu’, that the two words appear close together in the text and that the article was written in 2009, gives a high chance, though no guarantee, that its subject is the ‘Novel Influenza A (H1N1)’ [Wik] (also known as ‘swine flu’) that caused world-wide concern in 2009. Often, simplifying assumptions are needed on the theoretical formalisation of sophisticated retrieval methods, either for inherent lack of knowledge (e.g. it may be impossible to acquire certain statistical information on the data to search), or for performance reasons (the full theoretical formalisation may be computationally too intensive). Moreover, retrieval methods may need to be tuned specifically for the data they need to search.

Devising, formalising, validating and refining retrieval methods, together with a variety of other theoretical aspects (e.g. relevance measures, natural language processing, information extraction, see [vR79, MRS08] for an overview) are fascinating and demanding research topics that have traditionally dominated the scene in the field of information retrieval. Relatively little importance has been given to other aspects of the problem. One of these aspects is the *engineering* of search systems.

1.1 Problem definition

The traditional focus of information retrieval research on its most theoretical aspects is visible in the relatively poor design of search system architectures developed. For many years the scope of information retrieval has focused on relatively simple retrieval methods applied to text collections only. The development of search systems has mostly been regarded as the realisation of prototypes for the specific theoretical aspect being studied (or for a specific market segment in commercial settings), with a limited set of supported retrieval methods, data collections and parameters, that are hard-coded in the implementation. If on the one hand this pragmatic approach may result in efficient search engines for specific problems, on the other hand it implies low engineering efficiency: scarce modularity, low re-usability, high coding effort for each new search context to be addressed. As an example, consider the widely used search engine Lucene [HG04]. Next to the ranking algorithms it provides, new ones can be used, after they are integrated into the system. While this approach allows for some flexibility, a number of obstacles need to be addressed before this flexibility can be used in practice, ranging from the packaging of the new algorithm as a compatible Java class to the fact that the rather fixed set of statistics available in the system may not suffice[1].

Modern search systems are expected to cope with more diverse scenarios, accessing multiple and heterogeneous data sources and adapting to an increasing number of parametrisations. A structured engineering of search system architectures is required in order to deal with this increasing complexity. More specifically, the following critical aspects need to be addressed:

Content independence. Search systems tend to be highly specialised for specific content, retrieval methods and context parameters. Content independence [dV01, Mih06, WLMZ03], known as the decoupling between search strategies and logical content representation, is necessary in order to allow more flexibility.

Data independence. Search systems are built on top of physical data-access primitives (mainly inverted lists [ZM06]), which implies no (or limited) separation between search strategies and physical details such as algorithms and data models. Because data independence is one of the main achievements of database research, the idea of using database solutions as a physical interface for IR has been pursued for long (see [CRW05]) and resulted in a research field referred to as IR&DB integration. In spite of its intuitiveness, this integration turned out to be more difficult than expected, for two main reasons:

- efficiency: no DB implementations have so far provided sufficient run-time efficiency in comparison to custom-built IR solutions;

[1]A non-trivial integration of BM25/BM25F [RZ09] into Lucene is described in [PIPAFF09].

- data model mismatch: the typical set-based database interface is far from natural for retrieval method specification.

Abstraction layers. The lack of content and data independence in traditional search systems is due in the first place to the absence of proper abstraction layers that separate clearly the different stages of retrieval processes: the modelling of retrieval methods and content within a unified formalism; the declarative specification of retrieval methods; their implementation-independent and data-driven representation; their implementation-specific instantiation.

1.2 Approach and research questions

This thesis starts from the problem analysis summarised above and addresses the following general research question:

> *What are the requirements for a software architecture that provides search systems with* ***content independence****,* ***data independence*** *and* ***runtime efficiency and scalability****?*

Without loss of generality, we assume in the following a layered software architecture. We can now refine the question above into more specific **research questions**:

1. *Assuming a layered architecture, which abstraction layers should be distinguished, as a prerequisite for providing search systems with content and data independence?*

2. *What are suitable instantiations of such abstraction layers?*

3. *Can automatic mappings among the identified layers be defined, so that possible data model mismatches are taken care of by the system?*

4. *What are the requirements of such a software architecture in terms of runtime efficiency and scalability?*

1.3 Scientific contributions and thesis outline

The actual thesis promoted in this manuscript, based on the problem analysis and the research questions above, can be summarised in the following thesis statement:

> *IR tasks and concepts can be naturally modelled in terms of* ***matrix spaces*** *and implemented using the* ***array data model*** *as a conceptual data-access abstraction. This combination allows to enable content independence in a real search system architecture. The array data model can be mapped automatically onto the logical and physical data-access abstractions of the* ***relational data model****, and executed on a relational engine, which enables data independence. IR search tasks expressed and mapped this way can be deployed to instantiate a so-called* ***Parametrised Search System*** *(PSS). Such a layered architecture can compete with the efficiency and scalability of custom-built search systems, provided that:* (i) *mapping from the array to the relational domains is optimised with a specific set of* ***array-aware transformation rules****, with support for* ***sparse arrays****;* (ii) *a database engine tuned for* ***modern hardware exploitation*** *is employed.*

The scientific contributions that support the thesis statement above are addressed as outlined below.

Chapter 2: Related work. In this chapter, we put our work in perspective by offering a survey of previous research whose goals overlap with those presented in this thesis.

Chapter 3: Parametrised Search Systems. This chapter introduces the concept of a Parametrised Search System, which follows the design principles of a layered architecture. The resulting abstraction layers are the prerequisite for the implementation of search systems that support both a declarative language for their parametrisation and underling data independence capabilities. In the same chapter, the array data model is identified as a good candidate to support two layers at the same time: the IR modelling abstraction and the conceptual data-access layer. The work presented in this chapter addresses research *questions 1 and 2*, and has been partly published in [Cor06, CdV06, CdV07].

Chapter 4: The database approach to sparse array computations. This chapter proposes to support the declarative array data model identified in Chapter 3 with the benefits of the database approach, such as data independence and query optimisation. To make this proposition more concrete, we describe SRAM, a prototype mapping tool between the array and the relational domains. The work presented in this chapter addresses *research questions 2 and 3* and has been partly published in [Cor06, CdV06, CdV07, CHZ$^+$08].

Chapter 5: Array database optimisation. This chapter focuses on performance aspects of the array-database approach proposed. It describes several techniques for the optimisation of each of the query transformation steps throughout the abstraction layers of the system's architecture. Finally, it identifies the requirements a database engine needs to meet in order to provide sufficient execution speed for the proposed application stack. The work presented in this chapter addresses *research question 4* and has been partly published in [CvBdV04, vBCdV05, CHZ+08].

Chapter 6: Evaluation. This thesis proposes a solution to the aforementioned research problem and driving questions, which is summarised in the thesis statement above, and finally attempts to assess its quality. Ideally, the following questions can be used as quality indicators for this research:

- *How does an interface based on the array data model simplify the database-powered implementation of parametrised search systems?*
- *How flexible is the proposed 'array-database' approach with respect to different IR tasks and/or data collections?*
- *Is the performance provided by this approach satisfactory? What is the importance of an optimised automatic translation compared to the raw speed that the underlining database engine could provide?*

However, the nature of some of the goals being pursued makes it hard to evaluate solutions in such a way that these questions can be answered directly. While efficiency can be numerically measured (although unbiased comparisons may remain problematic), this is not the case for goals like engineering simplification and flexibility. In Chapter 6, we validate the proposed solution on a number of case studies that address the quality indicators above and provide useful insights for evaluating the quality of this research.

Chapter 7: Conclusions. In this chapter, we summarise the main scientific contributions and match those against the driving research questions. Finally, we outline possible future work directions.

The best ideas are common property.

- Lucius Annaeus Seneca,
Roman philosopher, statesman, dramatist.

2 Related work

With the explicit goal of pursuing a structured approach to the design and implementation of *search engine architectures*, one of the main research lines followed in this thesis addresses the long-standing issue of *IR&DB integration*. The IR&DB "gluing layer" identified in the solution proposed is based on the array data-model. While this three-way combination is a novelty, the array data-model is familiar to both the database and the information retrieval research communities: *array databases* use multi-dimensional array structures to support scientific applications and *multi-dimensional spaces* are often the most natural way to express *information retrieval* concepts. Aim of this chapter is to allow the reader to put in perspective the research presented in this thesis, by pointing at a selection of relevant prior work that is related to the research areas mentioned above.

Section 2.1 suggests key publications that explore the parallels between information retrieval concepts, multi-dimensional spaces and linear algebra. All these works can provide useful input for the implementation of information retrieval systems based on the array data-model. Section 2.2 deals with motivations, issues and possible approaches in supporting information retrieval with database technology. Finally, Section 2.3 gives an overview of research related to array databases.

2.1 Multi-dimensional spaces and information retrieval

Modelling of query terms and documents as vectors in high-dimensional spaces dates back to the SMART system [Sal71, SWY75, SM86], where Salton introduces the Vector Space Model (VSM) for information retrieval. Documents are represented as vectors in a space that has as many dimensions as unique terms in the collection to index. The value of each component in a vector reflects the importance of the associated term in the associated document. Typically, this value is a function of the frequency with which the term occurs in the document and/or in the collection and can be 0, which indicates no occurrence. Several different ways of computing these values, also known as (term) weights, have been developed. One of the best known schemes is tf-idf weighting [SWY75, SM86]. A collection that contains d documents and t terms is then represented as a matrix $DT[d, t]$. Similarly, a query can be seen as a pseudo-document and represented by a vector $Q[t]$, which indicates which query terms are considered important for the information need. Having defined document and query representations, the retrieval process consists of ranking documents by the result of a similarity measure between query and collection document. Given this geometrical interpretation, the most commonly used similarity measure is computed as the cosine of the angle between the query and document vectors (a cosine value of zero means that the query and document vector are orthogonal and have no match). Refer to e.g. [ZM98, MRS08] for more details and alternative similarity measures. One severe constraint of the original vector space model is that it does not allow to take term correlations into account easily. Terms are treated as a set of orthogonal vectors in the $DT[d, t]$ matrix, i.e. terms are assumed to be independent. Because of this, the VSM fails to handle correctly text collections where synonymy (different words with same meaning) and polysemy (words with more than one meaning) occur frequently.

A number of approaches have been proposed to overcome the limitations imposed by the term independence assumption. Wong et al. [WZW85] show how to compute such correlations automatically during the indexing process and how to include such information in the retrieval process, variant known as the Generalised Vector Space Model (GVSM). The Distributional Semantics based Information Retrieval (DSIR) model [Run88] focuses on using contexts to characterise the meaning of a word. Co-occurrence statistics are extracted from a document collection as a source of distributional information. The co-occurrence statistic of a word is the number of times that word co-occurs with one of its neighbours within a pre-defined boundary, the ‘distributional environment’, such as sentences, paragraphs, sections, whole documents, or windows of k words. Each term is then represented as a vector in the space of this co-occurrence matrix, and each document, in turn, as a vector of term vectors. Standard VSM similarity measures can be used in this augmented vector space to rank documents accordingly. Latent Semantic Indexing (LSI) [FDD$^+$88, DDL$^+$90, BF95, BDJ99] is based on the principle that words that are used in the same contexts tend to have similar meanings and performs a dimensional reduction of the term space, with the effect of preserving the most important semantic information in the text while reducing noise and other undesirable artifacts of the original matrix spaces. LSI uses a rank-reduced singular value decomposition (SVD) on a term-document matrix to identify patterns in the relationships between the terms and concepts contained in an unstructured collection of text. The rank-reduced SVD truncates the singular value matrix to size k, reducing the original term space into a k-dimensional space, with k much smaller (typically 300-400) than the total number of terms. As a general rule, fewer dimensions allow for broader comparisons of the concepts contained in a collection of text, while a higher number of dimensions enables more specific (or more relevant) comparisons of concepts. This dimensional reduction has two major benefits: the soft clustering of terms into concepts offers an automatic, language-independent, fully mathematical solution to retrieval quality issues coming from synonymy and polysemy; as a positive side-effect, collection indices become much smaller. Latent Dirichlet Allocation (LDA) [BNJ03, WC06] is a generative probabilistic variant of LSI, which assumes that words occurring in a collection can be generated from a set of concepts extracted from a small set of example documents. The Topic-based Vector Space Model (TVSM) [BK03] removes from the VSM the assumption of orthogonal term vectors, which allows for flexible specification of term-similarities and integrates more naturally stemming, stop-word lists and thesauri. A recent work by Melucci [Mel08] generalises vector spaces used in the VSM to allow retrieval in context, by making a parallel between a context and the basis of a vector: a vector, which represents an information object, is generated by a basis in the same way as the information object, or an information need, is generated in a context; every vector can be generated by a different basis in the same way as an information object is generated in different contexts. The probability of context, that is, the probability that an information-object has been materialised within a context, is a function of the distance between the subspace that a basis spans and an information-object vector.

Van Rijsbergen [vR04] shows how logical, probabilistic and vector space models can be combined in one mathematical framework, the very one used to formulate the general principles of quantum mechanics. Using this framework, van Rijsbergen presents a new theory for the foundations of IR, in particular a new theory of measurement. He shows how a document can be represented as a vector in Hilbert space, and the documents relevance by an Hermitian operator. All the usual quantum-mechanical notions, such as uncertainty, superposition and observable, have their IR-theoretic analogues. The standard theorems can be applied to address problems in IR, such as pseudo-relevance feedback, relevance feedback and ostensive retrieval. The relation with quantum computing is also examined. This framework is first used in practice by Piwowarski et al. in [PFLvR10].

Turtle, in his doctoral dissertation [Tur91], developed a probabilistic model for information retrieval formulated in terms of a Bayesian Network. Queries, concepts and documents are nodes in the graph representing an inference network. An edge between nodes a and b in this graph describes the conditional probability $P(a|b)$ of a causing b. For each node b, a *link matrix* encodes the probability of observing one of the m allowed values, given the values observed at b's parents (nodes reaching b). For n parents, the link matrix for b must encode m^{n+1} coefficients. Despite its name, this link matrix is really a multi-dimensional array that represents a tensor. Turtle, in his dissertation, shows how to compute efficiently its coefficients using tensor notation. Because tensors describe transitions in the inference network, they also describe operators, such as boolean AND, OR, NOT or weighted sum. These are the building blocks for the query language offered by the retrieval systems Inquery [CCH92] (now Indri [SMTC04]) and Galago [Gal, Str08], resulted from Turtle's research. Inquery's query language can be considered to enable some early version of a parametrised search system, because of the separation of concerns it allows between documents and content representation and has been used for precisely this flexibility (for example, in cross-language IR). However, it is not very flexible with respect to the retrieval model assumed.

The General Matrix Framework for Information Retrieval [RTK05] is a well-defined mathematical framework for modelling information retrieval, in which collections, documents and queries correspond to matrix spaces and key IR concepts are expressed with a set of standard linear algebra operations. The matrix spaces defined in this framework allow to express content-based retrieval (including some of the most popular retrieval models in the logical, probabilistic and vector space families), structure-based retrieval, semantic-based retrieval and evaluation. Finally, the interpretation of the eigenvectors of some content and structure matrices is discussed.

2.2 IR&DB integration

The information retrieval and database research fields evolved separately despite their obvious large overlap: essentially, they both study models and techniques to search in large amounts of complex information. Reasons for this divergence can be sought in the fundamentally different application areas that drove the development of the two disciplines. DB was driven by business applications, which emphasised technical aspects such as data consistency, precise and efficient query processing, deductive inference, and addressed users with high technical skills who write applications for exact-match searches using SQL, XQuery, etc. On the other hand, IR research was driven by applications for searching digital libraries, patent collections, etc. Technical requirements for this class of applications are more focused on best-effort processing, ranking and inductive inference, in order to maximise satisfaction of non-technical users with imprecise information needs. The result of this trend may be summarised today with a lack of proper ranking capabilities in the DB world and a lack of query optimisation in the IR world. Integration of IR and DB fields is acknowledged to be an important, still open, challenge [AAB$^+$05], especially in recent years, due to the increasing need for search applications that respond to both the driving criteria described above (mixtures of structured and unstructured data, as well as exact and approximate matches).

Motivations, issues and possible approaches for IR and DB integration are well summarised in a number of works. From an IR perspective, the need for more flexible systems with less engineering motivates such an integration, whereas from a DB view the goal is to embrace new ways of treating uncertain and imprecise data. Both the concepts of 'retrieval model independence' and 'content description independence' [Mih06] are described by Mihajlovic. Wen et al. use 'media independence' for a similar separation of concerns as Mihajlovic's content description independence [WLMZ03]. De Vries first defined the notion of 'content independence' [dV01] to refer to the decoupling between search strategies and content representation. A similar but more known concept is that of 'data independence', which refers to decoupling between search strategies and data primitives and structures. Already in 1996, Fuhr argued in favour of data independence in IR [Fuh96b], pointing out how this would reduce problems in plain text search with noun phrase search and treatment of compound words and (semi-)structured data types to capture attributes like author, journal title or publication year. A good summary of the DB-community view on integration with IR technology was presented during the SIGMOD 2005 panel [AYCR$^+$05]. In a SIGMOD 2007 keynote [Wei07], Weikum gives a chronological overview of IR&DB integration. In particular, he identifies the challenges of 2007 with those posed by XML IR and the future ones in the area of semantic web, an old dream that will, according to Weikum, re-gain attention thanks to the great progress in information extraction technology (see e.g. Extractiv [Ext]) and to the availability of "spontaneous annotations" created by the social web. In [CRW05], Chaudhuri et al. present a set of motivating examples, discuss different approaches of combining area-specific processing techniques, and propose an extension to the relational algebra that provides various IR features, e.g. a top-k operator, discussing the new challenges this algebra brings for the relational query optimisers.

Re-thinking the relational model to better support uncertainty. Important contributions to combined search of structured and unstructured data come from the research on probabilistic databases. Foundations of this approach include work by Schek and Pistor [SP82], who studied in 1982 non-first-normal-forms for IR problems as an alternative to standard relational algebra and Cavallo and Pittarelli [CP87], who introduced a first approach to probabilistic tuples. The PDM model developed later by Barbara et al. [BGMP92] uses nested relations to store imprecise attribute values, which overcomes some limitations of [CP87]. In the line of alternative data models, the AIM-P prototype [DL89, PD89] proposes radical changes, by supporting combinations of nested sets, records and sequences. The problem of imprecise attribute values has also been studied in terms of NULL values by the core database community in the seventies [Lip79, Vas79, IL84]. In particular, Codd [Cod79], with his three-valued logic (false,true,*maybe*), realises a simple form of ranking mechanism for database systems. Modern approaches to probabilistic databases build on the definition of a probabilistic relational algebra (PRA) and probabilistic datalog, by Fuhr and Rölleke [Fuh90, Fuh96a, FR97, Fuh00, RWWA08]. They describe evaluation strategies based on extensional semantics (efficient but not always correct) and intensional semantics (always correct, but more complex). Dalvi and Suciu [DS04] describe query evaluation on probabilistic databases in terms of a 'possible worlds' semantics and show how this is formally equivalent to the intensional semantics. They however discard the latter as an actual query evaluation method, due to its impractical complexity, and pursue a less generic approach. First they prove that certain classes of queries admit 'safe query plans', i.e. query plans that deliver correct answers when evaluated using extensional semantics. Then they describe an algorithm for safe query rewrites and approximations for those queries that do not allow safe query plans.

Taking state-of-the-art DB and IR solutions closer together. IR and DB research communities have made numerous attempts to apply lessons learnt from each other, with the explicit goal of minimising the impact on the respective pre-existing technologies.

First implementations of IR methods in database systems date back to the seventies, with e.g. the SEQUEL language [Mac79] and the work by Crawford [Cra81], who advocates the use of the relational model for IR. Stonebraker et al. [SAH87] proposed in 1987 a DB extension for text retrieval encapsulated in Abstract Data Types (ADT). This mechanism is relatively easy to implement, but has the disadvantage of remaining inaccessible to relational query optimisation. Today, we find this approach in loosely-coupled combinations of database systems and inverted file index structures, such as Sphinx [Sph]. Among the attempts to realise the inverse loose coupling, i.e. use a database system as a storage and index layer for an existing information retrieval engine, Inquery was integrated with an object-oriented database system in [CST92] and with the Mneme persistent object store, used to manage the Inquery's inverted file, in [BCCM93].

The idea to use DBMS technology as a building block in an IR system is pursued e.g. in [GFHR97], where the authors store inverted lists in a Microsoft SQLServer and use SQL queries for keyword search. Similarly, in [GBS04] IR data is distributed over a PC cluster, and an analysis of the impact of concurrent updates is provided.

Inverted files are the most widely used indexing structure in text search engines. In its simplest form, an inverted file is composed by a dictionary of all terms appearing in a collection and a set of inverted lists (or posting list) of document identifiers. Each term entry in the dictionary points to an inverted list of document identifiers in which the term occurs. [ZM06] offers an excellent overview on indexing techniques based in inverted files, including recent advances in terms of index compression, distribution and efficient resource utilisation [HBYFL92, MZ94, JO95, MMR00, Tro03, AM05b, MWZ06, MGC07, BB07]. Inverted files were also proven to always outperform signature files for text retrieval [ZMR98]. Various attempts were made to mimic the efficiency of inverted files in database systems and combine this with the data-oriented efficiency provided by query optimisation techniques, not only in text retrieval [Bla88, Put91], but also in XML retrieval [FKM00, ZND$^+$01, KKNR04] and multimedia retrieval [dV00]. Notable works on alternative text indexing approaches suitable for database systems include string B-Trees [FG99] and gram indices for exact substring matching on large text corpora [TSB09].

In 1985, Buckley and Lewit [BL85], in one of the earliest works on optimisation of retrieval based on inverted files, take a term-at-a-time approach to compute partial scores and decide when inverted lists need not be further examined. The state of the art on *top-k queries* over large (inverted) index lists has been defined in seminal work on variants of the threshold algorithm (TA) [Fag99, FLN01, GBK00, GBK01, NR99] [1]. Seamless combination of such

[1]Only in [TSW05] the connection between the TA algorithm and Buckley's work has been noticed.

IR-style top-k similarity queries with DB-style query processing and optimisation techniques is one of the open research challenges. Chaudhuri et al. [CDY95] studied query execution strategies with external text sources. Bruno et al. [BCG02] mapped top-k queries onto multidimensional range queries. Carey and Kossmann [CK97, CK98] introduced a stopping operator into pipelined query execution plans, supported by an extension to SQL which overloads the ORDER BY clause to request ranked results. Ciaccia et al. [CCG00] extended this work with generalised top queries, to allow per-group ranking of results. Blok et al. [BdVB99] and De Vries et al. [dVMNK02] describe, respectively, top-N and k-NN query optimisation techniques for multimedia databases that are designed to exploit a vertically decomposed storage model [CK85]. Theobald et al. investigate the computation of approximate top-k processing and evaluate this on both structured and unstructured data [TWS04] and describe how advanced variants of the TA algorithm can be used for efficient XML IR [TBM$^+$08]. Chaudhuri et al. [CDHW06] tackle the problem of returning the top-k matching tuples from conjunctive and range queries, by applying principles of probabilistic models inspired to information retrieval.

XML IR is what Weikum [Wei07] describes as the present challenge of IR&DB integration. Semi-structured data in XML format inherently call for IR-style flexible search functionalities combined with exact match queries. XQuery is the established equivalent of SQL in the XML world, i.e. its database query language; XQuery Full Text (XQFT) [AyS05, AYBC$^+$06] is an extension designed to incorporate text-matching, scoring and ranking functionalities. However, the highly flexible language currently defined in the W3C standard comes with semantic pitfalls [AYHR$^+$08] that are not addressed sufficiently to allow software developers to follow reliable guidelines. A number of XML IR systems have been developed nevertheless, supporting XQFT at their best, the much simpler NEXI (Narrowed Extended XPath) [RT04] language, or proprietary extensions. TopX [TBM$^+$08] supports both XQFT and NEXI and implements approximated TA-style top-k processing for semi-structured data. [KKNR04] studies the integration of structure indices for path traversal with inverted files for text ranking, with support for top-k queries based on the TA algorithm. Tijah [LMR$^+$05] is an XML retrieval system that follows the database design principles, by separating the conceptual (NEXI), logical (Scored Region Algebra, SRA [Mih06]) and physical (MonetDB [CWI]) layers. PF/Tijah [HRvOF06] extends Tijah by integrating it with the MonetDB/XQuery system [BGvK$^+$06], which enables seamless combination of database queries and IR ranking over XML data. The Staircase join [GvKT03] extends the set of physical operators available in MonetDB/XQuery to provide the database kernel with explicit support for XML tree-structures. XRank [GSBS03] proposes a ranking function for the XML "result trees", which combines the scores of the individual nodes of the result tree. The tree nodes are assigned PageRank-style scores [PBMW99] off-line. XCDSearch (also known as CXLEngine) [TE09, TE08] focuses on the importance of context-driven processing for loosely structured queries, which allow combinations of keywords and element labels. The notion of context is defined by parent-child XML node relationships.

In recent years *keyword search on relational DBMSs* has emerged as a simplified query approach to databases. Here, rather than DB technology supporting information retrieval applications, the goal is a non-structured database query-language that is more suited for user characteristics that are addressed in IR – low programming skills, partial and imprecise definition information needs. The main idea is to provide the user with a simple keyword search interface – the one familiar from web search engines – to access a database, without them having a detailed knowledge of the database schema or of the native query language. The system performs two main tasks (not necessarily in the presented order): it finds relevant tuples in the database with the help of standard IR techniques and explores the graph of connections (defined by both schema and data) existing among those tuples, to enable both result ranking and database browsing capabilities. The SIGMOD 2009 tutorial [CWLL09][2] offers an excellent overview of this fields of research. Some of the most relevant results include the implementations BANKS, Discover, DBXplorer, NUITS, Spark, and refinements of techniques therein used [HN02, HP02, ACD02, HGP03, LYMC06, WPZ$^+$06, LLWZ07, MYP09, QYC09, LFW09]

2.3 Array databases

The expression *array database* is used to indicate software for the management of data that are represented using multi-dimensional array structures. Array databases are commonly used to support scientific applications, such as for earth sciences, space sciences, life sciences, engineering and multimedia. Unfortunately, the overlap with general-purpose database technology is more limited in practice than the expression array database would suggest.

The concept of order in relational DBMSs, based on sets of tuples, is known during result presentation and data access optimisation. It is however irrelevant in the very core of a DBMS, the relational algebra. This is the key reason why

[2]Slides are currently available online: `http://www.cse.unsw.edu.au/~weiw/project/keyword_sigmod09_tutorial.pdf`

data structures such as lists and arrays are difficult to process natively in relational algebra, together with their inherent element orders. Maier and Vence, with "A Call to Order" [MV93] convinced the database research community in 1993 that this data-model mismatch is the primary cause for DBMSs failing to support adequately the development of scientific applications, often based on ordered data structures. An important milestone is reached by Libkin et al. in 1996 with the introduction of the functional array-language AQL [LMW96]. They show that inclusion of array support in their nested relational language entails the addition of two functions: an operator to produce aggregation functions and a generator for intervals of natural numbers. Sparse matrix operations are considered a query optimisation problem in [KPS97], confirming the potential of database technology for array processing. Here, however, the objective was to compile these into standalone programs, rather than to store and query sparse arrays in a DBMS. High re-usability of standard relational algebra and optimisation techniques, following the way paved by [LMW96], is the aim of the approach chosen in the dense-array mapping system RAM [vB09] that is used as a starting point for this thesis. Proposed as a general-purpose array mapping framework for scientific applications, [vB09] offers a detailed overview of the array data-model in programming languages, scientific software development and data management systems. More radical than [LMW96] is the approach of [Ng01], which proposes a partially ordered relational algebra (PORA) and an ordered SQL (OSQL) as a query language. The AQuery system [LS03], targeting financial stock analysis, uses the concept of "arrables", ordered relational tables, and an extension of SQL based on the clause `ASSUMING ORDER`. However, it only supports uni-dimensional arrays. The ADT/blob approach has been pursued in [HM04, How07], where an algebra for the manipulation of irregular topological structures, called gridfields, is applied to the natural science domain. The RasDaMan DBMS [BDF$^+$98, FB99] is a commercial array database system, implemented as an abstract data type in the O_2 object oriented DBMS [BDF$^+$98, FB99]. Its RasQL query language is a SQL/OQL like query language based on a low level array algebra, consisting of three basic operators: a tabulation-style array constructor (MARRAY), a condenser for computing aggregations (COND) and an index sorter (SORT). SciDB [CMKL$^+$09] and SciLens [Sci, KIML11] are currently the largest, ongoing, joint efforts towards the ambitious goal of realising an array database able to face most of the challenges encountered during all previous attempts. SciDB key features include a storage layer and algebra based on multi-dimensional arrays, bulk-oriented read/write operations, limited support for transactions, massive parallel execution, and built-in optimised operations for complex analytics. SciLens relies on the first-class efficiency of modern column-store databases, together with innovative (when applied to the relational world) approaches such as self-adaptive, approximate, incremental query answering, to introduce the necessary intelligence for handling massive amount of data.

Online analytical processing (OLAP) systems are based on the notion of *data-cubes*, structures that store data of interest over multiple dimensions, for an overview see [Vas98, VS99]. Data-cubes closely resemble multidimensional arrays. In [GB07], OLAP pre-aggregation techniques are formulated in terms of an array algebra and applied in raster image databases for geo services. The multi-dimensional array query model proposed in [Mac07] shares with this thesis the notion of shapes and index-based array operations, but focuses on more formal exploration of efficient bounds analysis and rule minimisation procedures of array queries. The ideas proposed in the RodentStore approach [CMWM09] aim at more flexible and efficient semi-automatic storage of complex data-structures and can contribute towards the definition of a full architecture for array databases. Rodentstore uses a declarative storage algebra whose expressions are interpreted to generate a physical representation of the data. Storage efficiency is also the focus of the RIOT (R with I/O Transparency) project [ZHY09]. Aim of the project is to extend R [R], an open-source statistical computing environment widely used by statisticians, to transparently provide efficient I/O. This is achieved by implementing highly efficient storage and I/O optimisation techniques and interface them with the language provided by the scientific application at hand (in the longer run, not limited to R), by means of an interpreted host language. Though the RIOT project targets for now a specific application domain, it also addresses the general problem of integrating database-style querying with programming languages. In particular, RIOT can interface with external database systems and make run-time decisions about the expected cost of (partially) pushing R operations into the DBMS and have them executed as relational operations versus moving part of data into the R system and process it internally. Its native storage manager is optimised to handle efficiently dense and sparse arrays, in a way that is similar to that described in this thesis; a linearisation function $f : \mathbb{N}^n \rightarrow \mathbb{N}$ maps n-dimensional array indices in the form $(i_1, \ldots, i_n)$ to scalars representing addresses inside a linear addressable storage container (e.g., a file on disk).

What would life be like if we had no courage to attempt anything?

- Vincent van Gogh, painter

3 Parametrised Search Systems

3.1 Introduction

For many years, information retrieval (IR) systems could have been adequately described as applications that assign an estimate of relevancy to a pair of document and query, each represented as a 'bag-of-words'. The implementation of such search systems has been relatively straightforward, and most engineers code the retrieval model directly on top of an inverted file structure.

Trends in research and industry motivate however a reconsideration of the above characterisation of IR. First, modern retrieval systems have become more complex, as they exploit far more than "just" the text. For example, the ranking function combines query and document text with other types of evidence, derived from, e.g., document markup, link structure or various types of 'context information' and includes filtering for spam or adult content. Also, work tasks supported by search have become diverse. Within organisations, *enterprise search* refers to intranet search, but also search over collections of e-mail, finding expertise, etc. [Haw04]. People use web search indeed for the goal of 'finding information about', but also to book a hotel, find a job, hunt for a house, just to name a few. Companies are targeting these niche markets with specialised search engines (referred to as *vertical search*).

Today, the development of such specialised applications is the job of information retrieval specialists. We expect however that, very soon, *any* software developer should be able to develop applications involving search. Actually, Hawking has stated that *"an obvious reason for poor enterprise search is that a high performing text retrieval algorithm developed in the laboratory cannot be applied without extensive engineering to the enterprise search problem, because of the complexity of typical enterprise information spaces"* [Haw04]. Simplifying the process of tailoring search to a specific work task and user context should therefore be an important goal of IR research.

3.2 Engineering of Search Systems

Modern IR application requirements challenge us to reconsider the design characteristics of search systems. We promote an innovation in the search system engineering process, by introducing more flexibility in the IR system's architecture. The increased flexibility aims to reduce the effort of adapting search functionalities to work task and user context.

This thesis proposes the engineering of search systems to proceed analogous to the development of office automation applications using relational database management systems: define the 'universe of discourse'; design a conceptual schema; express the user application in terms of this schema; and, design the user interface. So, software developers of a search application should have access to a high-level declarative language to specify collection resources *and retrieval model.* The search engine can then be parametrised for optimal effectiveness: adapted to the work task and user context, optimised for specific types of content in the collection and specialised to exploit domain knowledge.

3.3 Parametrised Search Systems

We refer to this new generation of information retrieval systems as *parametrised search systems* (PSS).

Fig. 3.1 illustrates how a PSS differs from a traditional search engine, in the so-called *abstraction layers*:

application interface - to express the user information need
(e.g. a simple keyword-list, a query-by-example interface, etc.)

application abstraction - to model and abstract the information need
(IR details are abstracted away)

IR modelling abstraction - to model an abstract IR task
(data-access details are abstracted away)

conceptual data-access layer - to express an IR task declaratively
(actual expression of a modelled IR task, logical/physical data-access details are abstracted away)

logical data-access layer - to express a data-access plan declaratively (instantiation of a conceptual data-access abstraction, physical details are abstracted away)

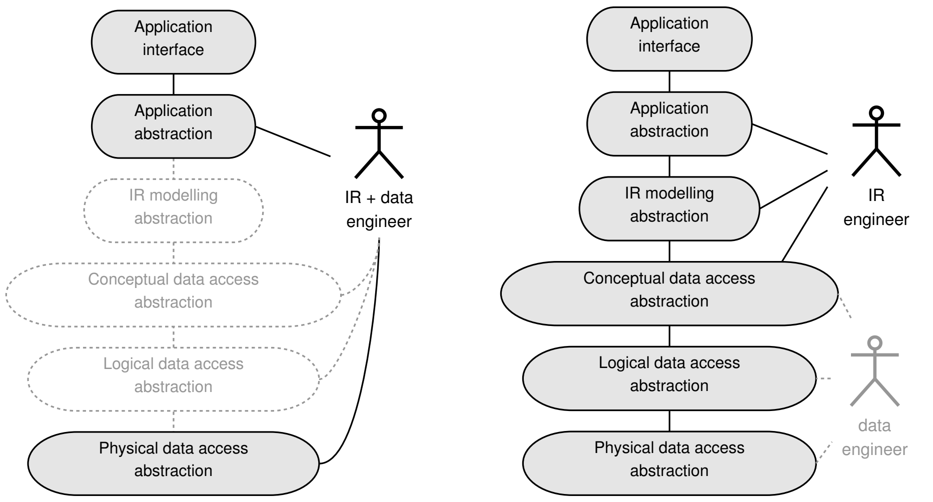
physical data-access layer - to express a physical data-access plan.

The reader may refer to Section 6.1 for an example of how each of the layers above can be instantiated in a real PSS.

Leaving un-specified the first two, application-related, layers, we focus in this chapter on identifying good candidates for the IR modelling abstraction and indicate possible choices for its data-access implementation. The layered approach to these two phases of the search, IR modelling and data access, is what really makes a PSS different from a traditional search system and takes the name of *parametrisation.* A PSS provides a declarative abstraction in which search application developers specify the desired search strategies with less effort. These specifications should stay as close as possible to the problem definition (i.e., the retrieval model to be used in the search application), abstracting away the details of the actual organisation of data and content. The system translates such specifications, customised for a specific user and task context, into operations on its internal data structures to perform the actual retrieval. The main rationale for this approach is to create an opportunity for decoupling search strategies from content representation, algorithms and data structures, enabling content and data independence in IR systems.

Compare this to current practice using IR software like Lucene [HG04], Lemur [OC01] or Terrier [OAP$^+$06]. These systems provide a variety of well-known ranking functions, implemented on top of the physical document representation. The modular design of the IR toolkit allows application developers to select one of these ranking functions suited for their search problem. Design decisions about e.g. how to rank – weighting various types of documents differently with respect to their expected contribution to relevancy – will however be part of the application code. Indri [SMTC04] and Galago [Gal, Str08] provide some more flexibility with their query languages based on Inquery [CCH92], even though the retrieval model specification remains not easily accessible.

The proposed layered architecture raises many open questions, the obvious ones of course what the IR modelling and data-access abstractions could look like. This thesis investigates a possible solution based on the *array data-model*, which is discussed in the remainder of this chapter, and its interaction with the *relational data-model*, which is the topic of chapters 4 and 5.


Application interface
Application abstraction
IR modelling abstraction
Conceptual data access abstraction
Logical data access abstraction
Physical data access abstraction
IR + data engineer
Application interface
Application abstraction
IR modelling abstraction
Conceptual data access abstraction
Logical data access abstraction
Physical data access abstraction
IR engineer
data engineer

(a) Traditional, IR and data engineering roles combined (worst case shown, intermediate cases in grey)

(b) Parametrised, IR and data engineering are two separate roles (possibly automated data engineering in grey)

Figure 3.1: Comparison of Search System architectures

3.4 An IR modelling abstraction

Many IR problems are naturally described in terms of multi-dimensional spaces [Sal71, SWY75, SM86, WZW85, MRS08]. The Matrix Framework for IR [RTK05] (abbreviated to Matrix Framework) is a mathematical formalism that maps a wide spectrum of IR concepts to matrix spaces and matrix operations, providing a convenient logical abstraction that facilitates the design and the analysis of IR models. Indexing, retrieval, relevance feedback and evaluation measures can all be described within the Matrix Framework. Also, it establishes a consistent notation for frequencies in event spaces, readily available as building blocks for IR applications in common matrix operation libraries. To see the Matrix Framework as an IR modelling abstraction in our desired PSS software architecture facilitates the engineering of extensible retrieval systems. It allows to express IR tasks declaratively, delegating physical details to a dedicated layer. At the same time, a matrix-based formalism is very well suited for being turned into concrete computational plans.

We introduce here the part of the Matrix Framework that is focused on indexing and retrieval. First, we define three vectors, one for each of the dimensions used in the framework: $D = [w_d]_{N\times 1}$ for *documents*, $T = [w_t]_{S\times 1}$ for *terms* and $L = [w_l]_{R\times 1}$ for *locations*, with $1 \leq d \leq N$, $1 \leq t \leq S$ and $1 \leq l \leq R$. The quantities $w_d \geq 0$, $w_t \geq 0$ and $w_l \geq 0$ are the weight of document d, term t and location l, respectively. In the simplest case, these weights are boolean values that denote presence/absence in the collection. The term "location" refers initially to term position in (linear) text documents, but can be generalised to cover concepts indicating document components of varying granularity, such as section, paragraph, position or XML element.

The content and the structure of each data collection are entirely described by the two boolean matrices $LD_{L\times D}$ (location-document) and $LT_{L\times T}$ (location-term), whereas each query is described by a vector $Q_{T\times 1}$. As defined in Equation (3.1) below, each value $LD(l,d)$ tells whether location l belongs to document d and each value $LT(l,t)$ encodes the occurrence of term t at location l of the collection. Finally, each query Q is described as a bit-vector with the length of the number of unique collection-terms.

$$LD(l,d) = \begin{cases} 0, & \text{if } l \notin d \\ 1, & \text{if } l \in d \end{cases}, \quad LT(l,t) = \begin{cases} 0, & \text{if } t \notin l \\ 1, & \text{if } t \in l \end{cases}, \quad Q(t) = \begin{cases} 0, & \text{if } t \notin Q \\ 1, & \text{if } t \in Q \end{cases} \tag{3.1}$$

Standard IR statistics are defined as simple operations on matrices LD and LT (the n in subscript should be read as 'number of'):

D_{nl} – **how many locations of a collection belong to a document?**

$$D_{nl} = LD^T \cdot L$$

T_{nl} – **how many locations a term occurs at, in the collection?**

$$T_{nl} = LT^T \cdot L \tag{3.2}$$

DT_{nl} – **how many locations a term occur at, in a document?**

$$DT_{nl} = LD^T \cdot LT$$

DT – **does a term occur in a document?**

$$DT = min(DT_{nl}, \mathbb{1}_{|D|\times|T|})$$

D_{nt} – **how many distinct terms appear in a document?**

$$D_{nt} = DT \cdot T$$

T_{nd} – **how many distinct documents contain a term?**

$$T_{nd} = DT^T \cdot D$$

where $\mathbb{1}_{m\times n}$ defines a matrix of size $m \times n$, filled with 1s, and $A \cdot B$ denotes matrix multiplication.

A number of standard IR frequencies are easily derived from the quantities defined above:

DT_f – **within-document term frequency**

$$DT_f(d,t) = \frac{DT_{nl}(d,t)}{D_{nl}(d)},$$

D_{tf}, D_{itf} – **term frequency and inverse term frequency of a specific document** d

$$D_{tf}(d) = \frac{D_{nt}(d)}{|T|}, \quad D_{itf}(d) = -logD_{tf}(d), \tag{3.3}$$

T_{df}, T_{idf} – **document frequency and inverse document frequency of a specific term** t

$$T_{df}(t) = \frac{T_{nd}(t)}{|D|}, \quad T_{idf}(t) = -logT_{df}(t),$$

T_f – **collection frequency of a term**

$$T_f(t) = \frac{T_{nl}(t)}{|L|}.$$

Finally, it is possible to define a number of retrieval models using the quantities and the frequencies described above. For example, *tf.idf*, Language Modelling and Okapi BM25, are shown below. Refer to [RTK05] for formal details on how these and other retrieval models are defined in the Matrix Framework.

tf.idf. The retrieval status value (RSV) of all documents against a query Q, following the basic *tf.idf* approach, is specified as:

$$RSV_{tf.idf} = DT_f \cdot \begin{pmatrix} T_{idf}(t_1) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & T_{idf}(t_{|T|}) \end{pmatrix}_{|T|\times|T|} \cdot Q$$

Language Modelling. The retrieval status value (RSV) of all documents against a query Q, following the Language Modelling approach, is specified as:

$$RSV_{LM} = \begin{pmatrix} \log(P(t_1|d_1)) & \cdots & \log(P(t_{|T|}|d_1)) \\ \vdots & \ddots & \vdots \\ \log(P(t_1|d_{|D|})) & \cdots & \log(P(t_{|T|}|d_{|D|})) \end{pmatrix} \cdot Q$$

$$P(t|d) = \lambda \cdot DT_f(d,t) + (1-\lambda) \cdot T_f(t)$$

Okapi BM25. The retrieval status value (RSV) of all documents against a query Q, following the Okapi BM25 approach, for given parameters k_1 and b, is specified as:

$$RSV_{BM25} = \begin{pmatrix} \omega(d_1, t_1) & \cdots & \omega(d_1, t_{|T|}) \\ \vdots & \ddots & \vdots \\ \omega(d_{|D|}, t_1) & \cdots & \omega(d_{|D|}, t_{|T|}) \end{pmatrix} \cdot Q$$

$$\omega(d,t) = T_{idf}(t) \cdot \frac{(k_1 + 1) \cdot DT_f(d,t)}{DT_f(d,t) + k_1 \cdot ((1-b) + b \cdot \frac{|D| \cdot |D|}{|L|})}$$

3.5 Data-access abstractions

The focus of the Matrix Framework for IR is to define an IR modelling abstraction; that is, to model IR concepts and tasks in mathematical terms, defining and manipulating multi-dimensional spaces, without leaving the theoretical domain. Shifting the focus towards implementation, a first point of concern is whether such a modelling abstraction is amenable to actual implementation in a software architecture, either directly or after some translation steps. This thesis proposes to use the Matrix Framework not just for modelling and analysis, but also as a declarative specification interface in actual IR system engineering. The concept of multi-dimensional spaces can be directly implemented by the *multidimensional array data-model*, available in a large variety of programming languages and scientific software.

One significant difference between the theoretical and the implementation-oriented points of view is that implementation cannot ignore concerns about feasibility and efficiency. For example, while size and value distribution of matrix spaces are irrelevant in the Matrix Framework, they are a crucial implementation issue. In particular, we should take advantage of the fact that most values of some matrices, such as LD and LT are equal to 0 (such matrices are commonly referred to as *sparse* matrices). Naive storage and manipulation of such matrices, when dealing with sufficiently large collections, would result in data structures impossible to handle, even with the hardware resources available nowadays. A first requirement is therefore that such mathematical spaces need optimised implementations that can store and manipulate efficiently sparsely distributed values.

The solution explored in this thesis is to support the identified IR modelling abstraction using array database technology. That is, to implement the Matrix Framework (IR modelling abstraction) with the array data-model (conceptual data-access abstraction), which is mapped onto the relational data-model and database technology (logical and physical data-access abstractions). This choice is motivated in the first place by the following observation: the ultimate goal of a search system, irrespective of the chosen system architecture, is to retrieve data that is relevant to users. The relational data-model and the database technology that support it are the obvious candidate building blocks for the implementation of logical and physical data-access layers. Despite its intuitiveness, this has to overcome a number of open issues in the topic of database and information retrieval integration [CRW05, AYCR$^+$05, Wei07]. Implementing linear algebra operations by means of database operations and hoping to compete with the performance of specialised numerical algorithms sounds admittedly like a hard challenge. A large number of software solutions can handle sparse multi-dimensional arrays efficiently. To mention one, Matlab [Mat] is a well-known numerical computing environment that provides array manipulation among other features and can be used as a computational back-end from custom-built applications. A more direct approach would implement array operations directly in the search application, by using highly optimised primitives such as the ones provided by the BLAS [LHKK79, bla02] development library. These packages are however rather limited with respect to data management facilities like access control and recovery, and they are usually not designed for scalability. Also, they focus on (and are optimised for) numeric data types only.

This thesis research advocates the benefits provided by database technology, such as query optimisation and data independence, over the importance of ultimate efficiency of basic array operators taken in isolation. Chapters 4 and 5 present the solution proposed for the automatic mapping of the array to the relational data-models, so that the concept of a PSS can be entirely implemented. Evaluation of the complete architecture and of some of its main aspects in particular (e.g. flexibility or efficiency) is the topic of Chapter 6.

Make everything as simple as possible,
but not simpler.
- Albert Einstein, physicist

4 The database approach to sparse array computations

Sparse array computation is widely addressed in numerical linear algebra, with a large amount of literature and many software implementations, in particular for two-dimensional arrays (matrices) [GL83]. Such solutions focus on the optimisation of single operations rather than full expressions (for example, many different implementations exist for matrix multiplication), assuming specific data encodings, or even specific hardware. The 'database approach' proposed in this thesis turns instead the numerical problem into a query optimisation problem, providing the following potential benefits:

data independence - relational expressions are transparent to the physical organisation of data. The data access optimisation problem is taken care of by a relational engine, rather than being bound to specific numerical algorithms.

resource utilisation - modern database engines are tuned for effective exploitation of the (possibly limited) hardware resources, and use cache-conscious, CPU-friendly algorithms, becoming more and more attractive for computational-intensive applications.

open 'black boxes' - instead of providing ad-hoc implementations of, say, matrix multiplication or matrix transposition algorithms, express them as a combination of native database primitives (join, selection, etc.). This allows the query optimisation process to take such operations into account within a larger problem and look for the overall best query plan.

Black boxes are pieces of software described in terms of their input, output and transfer characteristics without any knowledge of their internal workings.

Their correct usage in software architectures can bring important advantages, above all modularity and absence of side-effects. Their main disadvantage is that, given the impossibility to access their internal details, they are hard to adapt to work at their best in a context different from the one they were designed for. Specialised array-computation routines such as those in e.g. BLAS [LHKK79, bla02] are, from the point of view of an application using them, black boxes. Each operation has several optimised implementations, in order to cover a variety of matrix types in input (e.g. sparse/dense, small/large, clustered, triangular, etc.). For some applications this solution may be optimal. Especially in the context of data-intensive applications, such as search applications, black boxes of different granularity may however yield reduced engineering effort and better overall performance. Relational algebra forms the set of black boxes that this thesis proposes to use for sparse array computations.

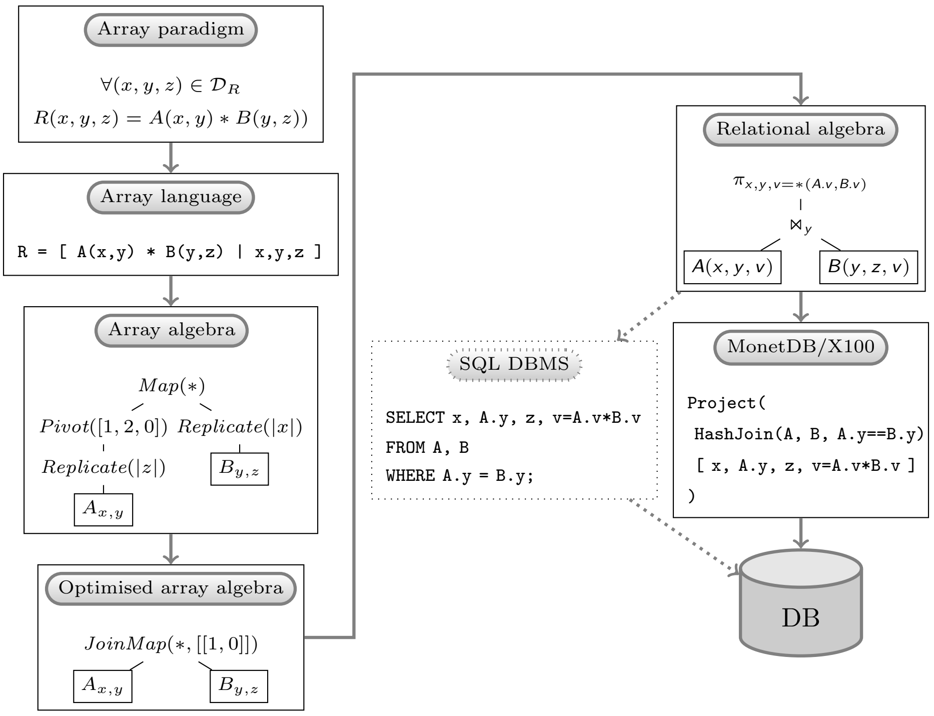
In a nutshell, while high-performance computing approaches problems with a strong focus on highly efficient and possibly highly distributed specialised algorithms, database technology focuses on query optimisation in search of the most efficient overall strategy to make large datasets flow through a sequence of algorithms. Both approaches have made attempts to learn from each other in the past. Kotlyar showed in [KPS97] how matrix multiplication can be solved efficiently when it is interpreted as a relational query optimisation problem that, given the characteristics of matrices in input, composes on the fly an algorithm based on relational primitives, with the aim of maximising properties as data-locality. On the other hand, recent database technology has evolved with more mature data processing capabilities, offering, next to the traditional optimisation of disk-based data access, careful design of CPU-friendly algorithms (see e.g. [Bon02, BZN05, ZBNH05]). While not that common, and leaving many open questions, these previous works support the feasibility of the investigations carried out in this thesis.

Now picture the following example scenario: an application is given a number of keywords that describe a book topic and has access to several online bookshops and social networks for user ratings; its task is to suggest the user books about the topic in input and highly rated by other readers, each with a pointer to the best online deal. This complex task requires a mixture of data retrieval (find titles, prices and ratings, join different sources on e.g. their ISBN) and information retrieval (rank books on their relevance to the topic in input) operations. Data retrieval operations naturally fall into the domain of database technology. Information retrieval operations can be described in terms of sparse array computations (see Chapter 3). The arguments presented above support an implementation of the latter based on relational black boxes. Suppose the data retrieval task outputs book items logically sorted by ISBN and grouped by edition, and physically stored by table columns rather than table rows. Will a subsequent sequence of non-relational operations be able to exploit such knowledge for best performance? Probably not, unless the available black boxes are break open and modified as needed. On the contrary, expressing the whole problem within the database domain allows a query optimiser to connect seamlessly the two different tasks and see them as one. Possible benefits include: identification and elimination of common sub-expressions, steering of the data-flow for optimal exploitation of data characteristics, such as ordering and grouping, which typically enable faster algorithms. Suppose now that a new algorithm for multi-threaded join proves to be faster than the existing ones. Its implementation in the database execution engine will immediately improve multi-threading capabilities and boost performance of both data and information retrieval tasks, which both use join operations extensively. Conversely, two different domains, one for data access and one for data processing, would force to maintain and improve two sets of efficient algorithms.

4.1 SRAM: Sparse Relational Array Mapping

The SRAM (Sparse Relational Array Mapping) system is a prototype tool that adds multidimensional array structures to existing database systems, by mapping arrays to relations and array operations to relational queries. The ability to interface to standard relational engines is a distinguishing aspect of SRAM, as most array oriented database efforts rely on proprietary data-structures. For example, the AQL language depends on a custom prototype back-end [LMW96], and languages such as AML [MS97], designed for image processing, and AQuery [LS03], designed for ordered data in the business domain, are realised by stand-alone applications. A notable exception is the RasDaMan system [Bau99], which does use proprietary data structures, but is implemented as an extension module for an object oriented database system. Storing arrays as relations instead of proprietary data structures allows the full spectrum of relational operations to be applied to the array data. This indirectly guarantees complete query and data-management functionalities, and, since the array extensions naturally blend in with existing database functionalities, the SRAM front-end can focus solely on problems inherent to the array domain.

SRAM builds on the RAM system [vB09], sharing many aspects of the user interface and the internal array algebra. Support for sparse arrays and for mapping to a relational algebra layer, together with the implementation of vari-



```
R = [ A(x,y) * B(y,z) | x,y,z ]
```

```
SELECT x, A.y, z, v=A.v*B.v
FROM A, B
WHERE A.y = B.y;
```

```
Project(
 HashJoin(A, B, A.y==B.y)
 [ x, A.y, z, v=A.v*B.v ]
)
```

Figure 4.1: Life-cycle of an array expression, from its mathematical formulation to its execution on a database (MonetDB/X100 query format is the default).

ous optimisation strategies, are however only available in SRAM. While SRAM syntax allows to express array operations on an element by element basis, the system translates such element-at-a-time operations to collection-oriented database queries, suited for (potentially more efficient) bulk processing. SRAM is RDBMS-independent, supporting translation to relational expressions. The experiments presented in this thesis are conducted using the database engine MonetDB/X100 [BZN05, ZBNH05], described in Section 5.4.1.

The life-cycle of array expressions through the SRAM architecture is depicted in Figure 4.1 and can be summarised by the following sequence of transformations:

1. Array paradigm $\mapsto$ Array language (Sections 4.2 and 4.3)
2. Array language $\mapsto$ Array algebra (Section 4.4)
3. Array algebra $\mapsto$ Relational algebra (Section 4.5)
4. Relational algebra $\mapsto$ Relational plan (Chapter 5)

Both array and relational algebra expressions are rewritten by a cost-based optimiser, the usual method in implementation of relational systems [Cha98]. This transformation process happens offline, i.e. without the necessity of a connection to a database engine that operates on actual data. This limits the availability of up to date statistics over the data and makes it impossible to apply run-time optimisation strategies, but allows to test the prototype and the quality of translations with reduced effort.

The next sections introduce the array paradigm, the array-syntax of the SRAM front-end, the array algebra layer, the relational algebra layer, and a set of mapping rules from array algebra to relational algebra.

4.2 Array paradigm: notation and definitions

Let us now introduces the notation for arrays used throughout this thesis. Section 4.2.1 includes general remarks on presentation aspects. Section 4.2.2 gives formal definitions of arrays and their properties.

4.2.1 General remarks on notation

This section may include terms and symbols that are defined formally only later on in Section 4.2.2 (e.g. "array" A, "default value" ε, etc.). Given the secondary role of such terms in the context presented, knowledge of such concepts is however not mandatory for a correct reading.

Let array names be capitalised and displayed in italic. The symbol A is used in most cases to indicate a generic array. When more arrays are involved, they are usually called B, C, and so on. When presenting real-life applications, longer and more descriptive names can be used, as $Scores$, $TopScores$, DT, etc.

Array names may appear as subscripts of other symbols. This indicates that the concepts expressed by such symbols are bound to the array in subscript, e.g.:

- $\mathcal{S}_A$: the shape of array A
- $\mathcal{T}_{Scores}$: the element type of array $Scores$
- ε_B: the default value of array B

When symbols appear with no array subscript, their relationship with a specific array is not known or not important. They are to be interpreted as a generic instantiation of the concept they represent, e.g.:

- $\mathcal{S}$: the shape of *an* array

- $\mathcal{T}$: the element type of *an* array
- ε: the default value of *an* array

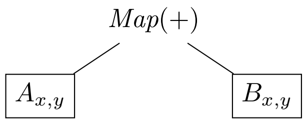
Matrices (2-dimensional arrays) are displayed in *row-major* order: their first and second axes refer to rows and columns respectively. For example,

$$A_{x,y}[2,3] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

is a matrix with 2 rows (axis x) and 3 columns (axis y).

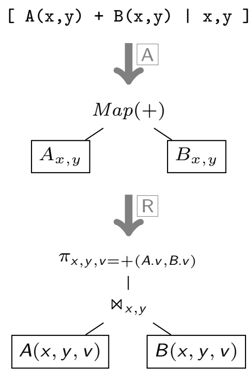
Examples often show how array computations are translated from one domain to another. We use a mono-space font for code snippets, e.g.: `[ A(x,y) + B(x,y) | x,y ]`. Array algebra expressions are displayed in italic, as mathematical formulae. Depending on their length, they may appear as normal text or be depicted graphically as a tree:

$Map(+, A, B) \equiv$



Likewise, relational expressions may appear as text or as trees. For relational tables and attributes, a Sans Serif font is used, as in $\mathsf{A.v}$.

Special arrows indicate translations from one domain to another: $\Downarrow\boxed{A}$ and $\Downarrow\boxed{R}$ map expressions to their array algebra and relational algebra equivalent, respectively. When a mapping is shown as in-lined text, the arrow $\mapsto$ may be used and the specific mapping domains are only implied through text fonts. For example, `Array` $\mapsto$ *Array* $\mapsto$ Array denotes an array mapped from array syntax onto its array algebra and relational algebra representations. The following example shows two subsequent translations, from array comprehension syntax to array algebra and finally to relational algebra.



```
[ A(x,y) + B(x,y) | x,y ]
```

Table 4.1: Notation used for arrays

Notation	Meaning
$\mathcal{S}_A \in \mathbb{N}_0^n$	shape (vector of dimension lengths) of A
$\mathcal{S}_A(i)$	length of the i-th dimension of A
$\mathcal{D}_A \subset \mathbb{N}_0^n$	index domain of array A : enumeration of indices of $\mathcal{S}_A$
$\bar{\imath} \in \mathbb{N}_0^n$	n-dimensional index vector $(i_0, \ldots, i_{n-1})$
$\mathcal{T}_A$	element type of array A
	(array type if A is nested, base type otherwise)
$(\mathcal{S}_A, \mathcal{T}_A)$	array type of array A
$\mathfrak{T}_A$	base type of array A
A	n-dimensional array $A : \mathcal{D}_A \to \mathcal{T}_A$
$(A_{i_0,\ldots,i_{n-1}})$	(with dimension variables $i_0, \ldots, i_{n-1}$)
$A[\mathcal{S}]$	n-dimensional array A of shape $\mathcal{S}$
$\|A\| = \prod_{i=0}^{n-1} \mathcal{S}_A(i)$	size of array A
$\|\mathcal{S}_A\|$	valence (number of dimensions) of array A
$A\,(\bar{\imath})$	array element indexed by $\bar{\imath}$
ε_A	default value of A
non-ε_A	a value $v \neq \varepsilon_A$
δ_A	density of A
c	a constant value c

To present translation patterns, *rule of inference* notation from formal logic is extensively used throughout this thesis. A rule of inference is a function which takes premises and returns a conclusion. The conclusion is then said to be inferable from the premises. In other words, a rule of inference can be read as "whenever in the course of some logical derivation the given premises have been obtained, the specified conclusion can be taken for granted as well":

$$\frac{\text{premise 1} \qquad \text{premise 2} \qquad \ldots}{\text{conclusion}}$$

A rule of inference is also called *transformation rule*, because it can be used to describe a transformation rule from a syntactic pattern (the premises) to a different syntactic pattern (the conclusion).

4.2.2 Definitions

An n-dimensional *array* A^n is defined as a function $A^n : \mathcal{D}_{A^n} \to \mathcal{T}_{A^n}$ that maps n-dimensional *array indices* to *array elements* of type $\mathcal{T}_{A^n}$. The array *domain* $\mathcal{D}_{A^n}$ is a set of n-dimensional array indices that forms a dense hyper-rectangle in $\mathbb{N}_0^n$. In other words, n-dimensional array indices are vectors of discrete non-negative numbers. Because the lower bound of $\mathcal{D}_{A^n}$ is chosen to be positioned at the origin of its hyperspace, a vector of axis lengths, called the *shape* $\mathcal{S}_{A^n}$, is sufficient to describe the bounding box of $\mathcal{D}_{A^n}$. Unless differently stated, arrays are assumed to be n-dimensional. In the remainder, the term n-dimensional can be omitted and the symbol A can be shorthand for A^n.

Because arrays are defined as functions, it is correct to read the array application $A(i_0, \ldots, i_{n-1})$ as "array A is applied to index $i_0, \ldots, i_{n-1}$". The more familiar "index $i_0, \ldots, i_{n-1}$ is applied to array A " is appropriate in the context of programming languages where arrays are defined as data structures.

Definition 4.1 (n-dimensional array shape). An n-dimensional array shape $\mathcal{S}_A$ is a vector of n integer values that represent the lengths of an n-dimensional array A's axes:

$$\mathcal{S}_A \in \mathbb{N}_0^n = (\mathcal{S}_A(0), \ldots, \mathcal{S}_A(n-1))$$

Square brackets can also be used, as in e.g. $[4, 2, 10]$. □

Definition 4.2 (n-dimensional array valence). The valence $|\mathcal{S}_A|$ of an n-dimensional array A is the number of its dimensions, equivalent to the length of its shape vector $\mathcal{S}_A$:

$$|\mathcal{S}_A| = |(\mathcal{S}_A(0), \ldots, \mathcal{S}_A(n-1))| = n$$

For example, the valence of atomic values, vectors and matrices is 0, 1 and 2, respectively. □

Definition 4.3 (n-dimensional array domain / index). An n-dimensional array domain $\mathcal{D}_A$ of array A is a set of n-dimensional indices that forms a dense hyper-rectangle in $\mathbb{N}_0^n$. $\mathcal{D}_A$ can be derived from $\mathcal{S}_A$:

$$\mathcal{D}_A = \{\bar{\imath}^n = (i_0, \ldots, i_{n-1}) \mid 0 \leq i_0 < \mathcal{S}_A(0), \ldots, 0 \leq i_{n-1} < \mathcal{S}_A(n-1)\}$$

An n-dimensional array index $\bar{\imath} = \bar{\imath}^n = (i_0, \ldots, i_{n-1})$ is a vector of discrete non-negative numbers. □

Definition 4.4 (element type). The symbol $\mathcal{T}_A$ indicates the type of A's elements. There are no restrictions on the element types that can be defined. Common types are for instance `bit`, `sht`, `int`, `lng`, `flt`, `dbl`, but more complex types as `temperature` or `color`, can be defined. Recursive types are also allowed, including an *array type*, which allows for the definition of *nested arrays*. Notice that tuple types, such as (`int` or `dbl`) are also allowed, which makes it possible to define multi-valued arrays (not used in this manuscript). □

Definition 4.5 (n-dimensional array). An n-dimensional array A is a function that maps n-dimensional indices $\bar{\imath} \in \mathcal{D}_A$ to elements $A(\bar{\imath}) \in \mathcal{T}_A$:

$$A : \mathcal{D}_A \to \mathcal{T}_A. \tag{4.1}$$

Note that the constraint that all elements in A are of the same type $\mathcal{T}_A$ follows directly from (4.1). Note also that atomic values can be thought as 0-dimensional arrays.
An array A whose shape is $\mathcal{S}$ can be denoted as $A[\mathcal{S}]$. □

Definition 4.6 (array type). Given an array A, its domain $\mathcal{D}_A$ can be derived from its shape $\mathcal{S}_A$. Therefore, the pair $(\mathcal{S}_A, \mathcal{T}_A)$ is a compact way to describe domain and range of the function $A : \mathcal{D}_A \to \mathcal{T}_A$. The entire set of possible array functions from a domain $\mathcal{D}$ to a range $\mathcal{T}$ is captured by the pair $(\mathcal{S}, \mathcal{T})$, called the array type. □

Definition 4.7 (n-dimensional nested array). An n-dimensional nested array A is an array whose elements are arrays themselves:

$$\forall \bar{\imath} \in \mathcal{D}_A : |\mathcal{S}_{A(\bar{\imath})}| > 0$$ □

Definition 4.8 (n-dimensional flat array). An n-dimensional flat array A is an array whose elements are atomic values:

$$\forall \bar{\imath} \in \mathcal{D}_A : |\mathcal{S}_{A(\bar{\imath})}| = 0$$ □

Definition 4.9 (base type). A base type $\mathfrak{T}$ is any type that is not an array type. The base type $\mathfrak{T}_A$ of array A is $\mathcal{T}_A$ if A is a flat array. If A is a nested array, its base type is the base type of its elements. □

Definition 4.10 (shape-aligned arrays). Arrays A and B are called shape-aligned or simply *aligned* when $\mathcal{S}_A = \mathcal{S}_B$. □

Definition 4.11 (n-dimensional sparse array). A sparse array A is an array in which a *default value*

$$\varepsilon_A \in \mathfrak{T}_A$$

occurs with a known or estimated frequency. Notice that this definition allows to choose arbitrarily an ε_A value (not necessarily the most common one), so long as its frequency in array A is known or can be estimated.

The *density* $\delta_A \in [0,1]$ of array A, which quantifies the portion of A that is not populated by ε_A values, is defined as:

$$\delta_A = 1 - \frac{\text{number of occurrences of } \varepsilon_A \text{ in } A}{|A|}.$$ □

Definition 4.12 (n-dimensional dense array). An array A is called dense when no default value is defined for it. The density δ_A of a dense array is defined as equal to 1. □

Sparsity is not an intrinsic property of arrays as mathematical entities. Instead, declaring arrays as sparse or dense is an arbitrary decision, that makes the computational framework in which they are used aware of such a property.

Example 4.1 (Sparse and dense arrays).
Consider the following array:

$$A[4,4] = \begin{bmatrix} 2 & 2 & 2 & 4 \\ 2 & 0 & 2 & 2 \\ 2 & 2 & 0 & 2 \\ 2 & 2 & 2 & 0 \end{bmatrix}$$

As such, it cannot be called sparse nor dense. When we intend to use it in a computational framework as SRAM however, we may want to enforce its sparsity properties.

A number of options are possible:

- array A is sparse with $\varepsilon_A = 2$ and therefore $\delta_A = 1 - \frac{12}{16} = 0.25$.
- array A is sparse with $\varepsilon_A = 0$ and therefore $\delta_A = 1 - \frac{3}{16} = 0.8125$.
- array A is sparse with $\varepsilon_A = 4$ and therefore $\delta_A = 1 - \frac{1}{16} = 0.9375$.
- array A is dense, which implies $\delta_A = 1$
- array A is sparse with $\varepsilon_A = 5$, even though value 5 never occurs in A. This yields $\delta_A = 1 - \frac{0}{16} = 1$. □

As we will see later on, especially in Chapter 5, the definition of the sparsity property of an array has a performance impact on the evaluation of expressions involving such an array. In general, this impact is more positive as the density δ_A decreases, but can be negative when δ_A increases over a certain threshold.

Table 4.1 summarises the notation used for arrays throughout this thesis.

```
/*****************************
* SRAM script: bestCities.ram
****************************/

// standard base type and function declarations
#include "types.ram"
// some application-specific constants and macros
#include "cityAppeal.ram"

// Cities is an explicit array of strings
Cities = [ "Amsterdam", "Madrid", "Moscow", "Rome", "Tokyo" ]

/* CityTemp and CityRainCm are stored nested arrays that contain,
   for each city, one-year data on temperature and rain falls */
CityTemp   = ([5],([365,24], flt)) "Table_CityTemp"
CityRainCm = ([5],([365,24], int)) sparse(0, 0.1) "Table_CityRainCm"

// let's define a vector of "city appeal factors"
CityAppeal = [ cityAppeal(CityTemp(city), CityRainCm(city)) | city ]

// find the id of the 2 cities with highest appeal factor
BestIdx = topN(CityAppeal, 2, Desc)

// finally, store the name of these "best" cities
BestCities := [ Cities(BestIdx(x)) | x ]
```

Figure 4.2: An example SRAM script: `bestCities.ram`

4.3 A language for sparse array computations

The SRAM language allows to read/write from/to persistent storage, create and manipulate arrays as defined in Section 4.2. This section introduces its syntax.

4.3.1 Structure of SRAM scripts

SRAM scripts are composed by language constructs that are oriented at array operations. Figure 4.2 shows a toy SRAM script that is used throughout this section as a running example. It reads weather statistics related to a number of cities from persistent storage, and determines the best cities where to live (weather-wise). Although this script is trivial in many respects, it shows most of the main syntax elements.

SRAM scripts can contain *empty lines*, *comments*, *statements* and *loop constructs*.

Comments

Single-line as well as multi-line comments are allowed:

```
/****************************
* SRAM script: cityAppeal.ram
****************************/

// alpha is a constant
alpha = 0.2

/* city appeal factor is computed as a function
   of temperature and rain falls */
cityAppeal(Temp,RainCm) = alpha * avg(Temp) / avg(RainCm)
```

Figure 4.3: `cityAppeal.ram`

Single-line comment (e.g. Figure 4.2, Line 5)

`// this is a comment`

Multi-line comment (e.g. Figure 4.2, lines 1-3)

```
/* this is
 a comment */
```

Statements

Each line of a SRAM script belongs to one of the following statements:

Include directive (e.g. Figure 4.2, Line 6)

`#include <filename>`

Inserts the content of an external SRAM script at the current position.

Declaration of a base type (e.g. Figure 4.4, Line 6)

`<base type>`

Declares `<base type>` as available in the system.

Declaration of a function prototype (e.g. Figure 4.4, Line 15)

`<function name> (<base type>, <base type>, ...)  <base type>`

Declares a function prototype: name, input types and output type. Such a function is thereafter available in the system and can be applied to atomic elements with the correct base type. Its implementation is expected to be available in the execution engine.

Declaration of an aggregation function prototype

(e.g. Figure 4.4, Line 25)

`<function name> {<base type>} <base type>`

Declares an aggregation function prototype: name, input type and output type. Such a function is thereafter available in the system and can be applied to arrays with the correct base type. Its implementation is expected to be available in the execution engine.

Declaration of a persistent array (e.g. Figure 4.2, Line 15)

`<array name> = <persistent array>`

Creates an array variable `<array name>` and binds it to a persistently stored array.

Persistent storage of an array (e.g. Figure 4.2, Line 25)

`<array name> := <array constructor>`

Creates an array variable `<array name>`, binds it to `<array constructor>` and stores it persistently

Deletion of an array from persistent storage

`delete <array name>`

Makes the array associated to the variable `<array name>` no longer persistently stored.

Macro definition (e.g. Figure 4.3, lines 6 and 10)

`<macro name>(x,y,z,...)  = <macro body>(x,y,z,...)`

Defines a parametric macro that is symbolically expanded throughout the rest of the script.

Loop constructs

Loop constructs are available with limited functionality. Because relational algebra has no means to express loop constructs, the only way to implement them at the SRAM syntax level is to unroll loops by explicitly replicating their body the specified number of times. Besides the growth of the SRAM script to process and of the resulting relational expressions, which can pose efficiency issues, the main disadvantage of such an implementation of loop constructs is that only fixed-length loops can be expressed. This is a limitation that prevents to express e.g. iterative algorithms whose stop condition is dynamically based on the data itself, rather than on a pre-defined number of steps. Two forms of loop construct are available:

Without control variable

```
repeat(n)
 <loop body>
end repeat
```

replicates `<loop body>` n times.

With control variable

```
repeat(x,n)
 <loop body>
end repeat
```

This second construct makes available the integer variable `x` that can be used to identify the current step in the iteration, in case the expressions in `<loop body>` need such a knowledge. This variable, however, cannot be used to quit the loop before the specified number of steps n.

4.3.2 Naming conventions and scope of syntax elements

Within each statement, the SRAM language allows the following syntax elements:

Literals

(e.g. Figure 4.2: `"Amsterdam"`, Figure 4.3: `0.2`)
Text strings (between quotes), numerals, booleans, etc.

Array identifiers

(e.g. Figure 4.2: `Cities`, `CityTemp`, `CityRainCm`, `BestIdx`)
Naming convention: they start with an uppercase letter.
Scope: global, downward.

Axis identifiers

(e.g. Figure 4.2: `city`, `x`)
Naming convention: they start with a lowercase letter.
Scope: local to the array definition they appear in.

Function identifiers

(e.g. Figure 4.3: `*`, `/`)
Naming convention: they start with a lowercase letter or a non-numeric symbol. They cannot be 0-ary functions. Infix notation is allowed for binary functions, (e.g. `2 + 3`), parentheses and comma-separated arguments, must be used for other functions (e.g. `log(2.0)`).
Scope: global, downward.

Base type identifiers

(e.g. Figure 4.4: `bit`, `str`, `int`, etc.)
Naming convention: they start with a lowercase letter.
Scope: global, downward.

Parametric macro identifiers

(e.g. Figure 4.3: `cityAppeal`)
Naming convention: they start with a lowercase letter.
Scope: global, downward.

Macro parameter identifiers

(e.g. Figure 4.3: `cityAppeal(Temp,RainCm)`)
Naming convention: macro parameter identifiers can represent anything that is allowed in the syntax, thus no specific naming convention for them.
Scope: local to the statement they appear in.

Constant (non-parametric macro) identifiers

(e.g. Figure 4.3: `alpha`)
Naming convention: start with a lowercase letter.
Scope: global, downward.

A simple example may clarify a few important points:

```
A = ([10,5], int) "Table_A"

mxTrnsp(A) = [ A(j,i) | i,j ]

[ log(mxTrnsp(A)(i,j)) + 2.0  | i,j ]
```

The symbol `A` in the first expression is an array identifier and its definition has global, downward scope. The symbol `A` in the third expression refers to that definition. The symbol `A` in the second expression, however, is a macro parameter identifier and its scope is local to that statement: it has no relation with the definition in the first expression (to avoid confusion, such name conflicts should be avoided when possible). Similarly, axis identifiers `i` and `j` are only valid within the statement the appear in: those in the second expression have no relation with those in the third expression.

4.3.3 Type system

The SRAM type system is extensible, in that *base types*, *functions* and *aggregation functions* are completely user-defined. This is possible because SRAM type checking is static (i.e. it happens at compile-time) and because base types and functions are mapped onto the final relational query language without interpretation steps. The only exception is the base type `int`, which is always defined, as it is used by the system for array indices.

Figure 4.4 shows an excerpt of type and function declarations. The following three sections detail on such declarations and may refer to specific lines of Figure 4.4.

Base type declarations

Base types are declared by simply listing them, as in Line 6. This enables the definition of subsequent arrays and functions that use such types as base types.

Function declarations

Function prototype declarations follow the pattern:

```
<function name> (<base type>, <base type>, ...) <base type>
```

For example, Line 15 declares the prototype of a binary function `*`, whose first and second input atomic values must be of types `flt` and `int` respectively, and whose output atomic value is of type `flt`. This line ensures that the system is able to statically derive the base type of any expression that contains function `*`. Note that base types `flt` and `int` must have been previously declared.

Cast function declarations define the prototype of functions that cast an atomic value from one base type to another. Although they play a specific role in the programming environment, technically they are no different from any other unary function. An example cast function declaration is shown at Line 20: a function called `flt` takes an atomic value of type `int` and returns an atomic value of type `flt`. Note that, although by convention the name of cast functions matches their return base type, this is not mandatory: `"cast_to_flt" (int) flt` would be also possible, provided that such a function name is finally mapped correctly onto a function name supported by the relational engine.

Aggregation function declarations

Aggregation functions are special functions that take arrays and return atomic values. Their declaration follows the pattern:

```
<function name> {<base type>} <base type>
```

Notice the syntactic difference with the declaration of standard functions: the expression in curly brackets, `{<base type>}`, indicates a bag of values of the same type in input. Line 25 shows a typical example of aggregation function, `sum`. Once aggregation functions are declared, they can be used in expressions, as explained in Section 4.3.7.

```
/*****************************
* SRAM script: types.ram
*****************************/

// list available base types
bit,str,int,flt,dbl,color,url

// list available functions (random excerpt)
"log" (dbl)     dbl
"pow" (dbl,dbl) dbl
"<"   (int,int) bit
"=="  (int,int) bit
"and" (bit,bit) bit
"+"   (int,int) int
"*"   (flt,int) flt
"/"   (dbl,dbl) dbl
// ...

// list available cast functions (random excerpt)
"flt" (int) flt
"dbl" (flt) dbl
// ...

// list available aggregation functions (random excerpt)
"sum"   {dbl} dbl
"avg"   {dbl} dbl
// ...
```

Figure 4.4: `types.ram`

4.3.4 Declaration of persistent arrays

Arrays must be described in the system before they can be read from persistent storage and used. In alternative to a fully fledged system catalogue, a data dictionary can be made available to the system at run-time by declaring arrays explicitly in each SRAM script. A persistent array declaration binds an array variable name to the following information:

- An array type $(\mathcal{S}, \mathcal{T})$ (see Section 4.6)
- Sparsity information (ε, δ) (see sections 4.11 and 4.12)
- The name of the relational table(s) where the array is stored

In SRAM syntax, the declaration of a persistent array follows this pattern:

```
<array name> = <array type> <sparsity> <storage specification>
```

Figure 4.2 shows the declaration of both a dense and a sparse array. The dense array *CityTemp* is declared at Line 15. Notice that an empty `<sparsity>` specification is synonymous of `dense`. The `<array type>` `([5],([365,24], flt))` declares a nested array of type ($\mathcal{S}_{CityTemp} = [5], \mathcal{T}_{CityTemp} = ([365, 24], \texttt{flt})$). The inner array's type is ($\mathcal{S} = [365, 24], \mathcal{T} = \texttt{flt}$), which can host temperatures (expressed as float values) for 365 days a year, 24 hours a day. The base type of the nested array *CityTemp* is therefore $\mathfrak{T}_{CityTemp} = \texttt{flt}$.

The sparse array *CityRainCm* is declared at Line 16. Here, the array sparsity is defined as `sparse(0, 0.1)`, which informs the system that the array is to be considered sparse, with $\varepsilon = 0$ and $\delta = 0.1$. The `<array type>` `([5],([365,24], int))` is similar to the one of *CityTemp*, except that the base type of the nested array *CityRainCm* is $\mathfrak{T}_{CityTemp} = \texttt{int}$.

Note that `<storage specification>` can be used in combination with both the widely used N-ary storage model (NSM) and the Decomposed Storage Model (DSM) [CK85, BK99, SAB$^+$05]. With an NSM-oriented table representation, `<storage specification>` is a single table name, as in:

```
A = ([10,20], int) "Table_A"
```

With a DSM-oriented table representation, used in column-store database engines, all index and value columns are addressed independently, as in:

```
A = ([10,20], int) (["i0_A", "i1_A"], "v0_A")
```

4.3.5 Persistent storage of arrays

Arrays can be saved to persistent storage for later usage, using the array variable assignment symbol '`:=`', as in the following syntax:

```
<array name> := <array constructor>
```

This creates an array variable `<array name>`, binds it to the result of `<array constructor>` and stores it persistently. An example of array storage is given in Figure 4.2 at Line 25: the array resulting from the expression `[ Cities(BestIdx(x)) | x ]` is associated to the array variable `BestCities` and stored with that name.

Note that a similar pattern

```
<array name> = <array constructor>
```

where the symbol '`=`' is used instead of '`:=`', is a *macro definition* (see 4.3.8), which does not store the array persistently.

4.3.6 Array construction

Within each statement, different array constructors can appear. These are expressions whose result is an array. Array construction is side effect-free and cannot be used to define how to modify an existing array in-place: it always defines shape and content of a new array (this aspect is further analysed in Section 4.4.1)

Different types of array constructor are available: comprehension syntax, explicit syntax, array element extraction, Sort, TopN, Concat, If-Then-Else. Hereafter they are discussed in detail.

Comprehension syntax

Comprehension syntax [BLS+94] is the core of SRAM language. It allows to declare arrays by means of the following construct:

$$[\ f(\mathtt{i}_0, \ldots,\ \mathtt{i}_{n-1})\ \mid\ \mathtt{i}_0\mathtt{<}\mathcal{S}(0), \ldots, \mathtt{i}_{n-1}\mathtt{<}\mathcal{S}(n-1)]$$

or, in a more compact form:

$$[\ f(\bar{\imath})\ \mid\ \bar{\imath}\mathtt{<}\mathcal{S}\ ].$$

The part denoted by $\mathtt{i}_0\mathtt{<}\mathcal{S}(0), \ldots, \mathtt{i}_{n-1}\mathtt{<}\mathcal{S}(n-1)$ specifies the new array's shape $\mathcal{S}$ (see Definition 4.1), namely the number of dimensions and the length of each dimension, which in turn defines the array index domain $\mathcal{D}$ (see Definition 4.3). Note that $\bar{\imath}\mathtt{<}\mathcal{S}$ corresponds to $\bar{\imath} \in \mathcal{D}$. The value of each dimension variable $\mathtt{i}_j$ ranges from 0 to $(\mathcal{S}(j) - 1)$. Dimension variables must be specified and named.

The part denoted by $f(\mathtt{i}_0, \ldots,\ \mathtt{i}_{n-1})$ maps each n-dimensional index $\bar{\imath} = (i_0, \ldots, i_{n-1}) \in \mathcal{D}$ to a value, by applying a function f. Recall from Definition 4.5 how arrays are formally defined: an array A is a function from its domain index $\mathcal{D}_A$ to its array type $\mathcal{T}_A$. It is important to note how comprehension syntax matches this definition: a comprehension describes such a function $f : \mathcal{D}_A \mapsto \mathcal{T}_A$, by explicitly defining the mapping from each $\bar{\imath} \in \mathcal{D}_A$ to the corresponding array element $A(\bar{\imath}) \in \mathcal{T}_A = f(\bar{\imath})$.

Example 4.2 (Comprehension syntax).
The expression

```
    B = [ log(A(y,x)) | x<5, y<6 ]
```

defines a new array $B_{x,y}$ with $\mathcal{S}_B = [5, 6]$, where the value of each element $B(x, y)$ is computed by applying the function log to element $A(y, x)$ of array A. Element $A(y, x)$ is in turn obtained by applying function A to indices y, x. □

Nested comprehension syntax

Recall from Section 4.2 that array elements can be arrays themselves, which yields nested arrays. This is supported in comprehension syntax by allowing $f(\bar{\imath})$ to be, in turn, a comprehension syntax:

$$\texttt{[<comprehension syntax> | } \bar{\imath}\texttt{<}\mathcal{S}\texttt{]}$$

Example 4.3 (Nested comprehension syntax).
The expression

```
B = [ [ log(A(y,x)) + z | z < 10 ] | x<5, y<6 ]
```

defines a new array $B_{x,y}$ with $\mathcal{S}_B = [5,6]$ and $\mathcal{T}_B = ([10], \texttt{dbl})$. The value of each element $B(x,y)$ is an array with a single axis named z, shape $\mathcal{S} = [10]$ and type $\mathcal{T} = \texttt{dbl}$. The value of each element $B(x,y)(z)$ of such an array of arrays is computed by the expression $\log(A(y,x)) + z$), which determines a base type $\mathfrak{T}_B = \texttt{dbl}$. □

Comprehension syntax with implicit axis domain

The $\bar{\imath}\texttt{<}\mathcal{S}$ part of a comprehension syntax may contain axes with *implicit axis domain*. For those axes, only the name is given, while the length is omitted as it can be inferred automatically by the system. This possibility is particularly important for the definition of useful *macros* (see Section 4.3.8).

For each axis with implicit domain, the largest possible domain is chosen, such that it is contained in the index domain of all the arrays using that axis name. This means that in case the axis is used to index more than one array, and different index domains are defined for that axis, the intersection of those domains is chosen. In case of empty intersection, the implicit domain for these axes cannot be computed.

Example 4.4 (Implicit axis domain).
The expression

```
A = ([6,5], dbl) "table_A"
B = [ log(A(y,x)) | x, y ]
```

is equivalent to the following:

```
A = ([6,5], dbl) "table_A"
B = [ log(A(y,x)) | x<5, y<6 ]
```

In the first expression, the domains of axes x and y of array B are assumed to be equal to the domains of the second and first axes of array A respectively. Hence, $B_{x,y}[5,6]$ is derived automatically.

Result shapes with partially implicit domains are also allowed:

```
A = ([6,5], dbl) "table_A"
B = [ log(A(y,x)) | x<5, y ]
```

or

```
A = ([6,5], dbl) "table_A"
B = [ log(A(y,x)) | x, y<6 ]
```

Again, the shape $[5,6]$ is derived automatically for array B. In the first case, axis y is bound to the first axis of array A, whereas in the second case axis x is bound to the second axis of array A.

Finally, we see an example that deals with conflicting index domains:

```
A = ([6,5], dbl) "table_A"
B = ([3], dbl)   "table_B"
C = [ log(A(y,x)) + B(x) | x, y<6 ]
```

The result of this expression is an array $C_{x,y}[3,6]$. Axis x is used to index both arrays A and B. Its length is 5 in A's index domain, while it is 3 in B's index domain. Intersecting the two corresponding index domains yields a length 3 for axis x: as large as possible but small enough not to exceed the index domain of both A and B. □

Figure 4.2 shows additional examples of comprehension syntax with implicit axis domain, at lines 19 and 25. At Line 19, the length of axis `city` is inferred as 5, as this axis is used to index arrays `CityTemp` and `CityRainCm`, both with shape [5]. At Line 25, the length of axis `x` is inferred as 2, as this axis is used to index array `BestIdx`, whose shape is [2].

Explicit syntax

A second constructor enumerates all the array elements explicitly. The definition of a one-dimensional explicit array of shape $\mathcal{S} = [5]$ looks like this:

```
[ 10 42 0 3 1 ]
```

Explicit arrays of higher dimensionality can be specified by array nesting. The following expression defines the explicit array of type $([2], ([5], \texttt{int}))$:

```
[ [10 42 0 3 1] [7 4 19 5 6] ]
```

In Figure 4.2, at Line 11, array `Cities` is defined using explicit syntax. Its shape is $\mathcal{S}_{Cities} = [5]$ and its element type is $\mathcal{T}_{Cities} = \texttt{str}$.

Array element extraction

Array element extraction is the result of the application of an array, which is a function, to an array index: given an array A and an array index $\bar{\imath}$, the application $A(\bar{\imath})$ performs an array element extraction operation. When element extraction is performed on a nested array, the result is again an array. This makes it fall in the category of array construction operations.

Figure 4.2 shows an example at Line 19: in the expression `[ cityAppeal(CityTemp(city), CityRainCm(city)) | city ]` two arrays of shape $[365, 24]$ are extracted from `CityTemp` and `CityRainCm` respectively, for each value of `city`.

Sorting

Sorting array values is allowed for one-dimensional arrays only. The resulting array does not contain sorted array element values, but indices of the sorted values in the original array, which would otherwise be lost during sorting.

The following construct returns a *dense* array S, with $\mathcal{S}_S = \mathcal{S}_A$ and $\mathcal{T}_S =$ `int`:

```
S := sort(A, <ASC|DESC>)
```

The actual values can subsequently be fetched by element extraction from the original array, using indices stored in array S:

```
[ A(S(x)) | x ]
```

Notice that the domain of axis variable `x` can be implicit, as it is used to index array S, whose shape is known as $\mathcal{S}_A$.

Sorting multi-dimensional arrays along one or more dimensions is a well-defined operation in the array domain (e.g. sort each column of a matrix separately). The restriction to one-dimensional arrays is due to the lack of a standard relational operation that would perform a “grouped-sort”, i.e. which would sort tuples within each group (in this case, groups would be identified by all dimension attributes but the one on which the sort is performed). Although a work-around is possible by using sorting as an expensive means for grouping, this remains a fundamental limitation in the relational algebra. The problem is identified already in [CCG00], where per-group top-N processing is studied.

Retrieving top-N values

Retrieving the *top-N* values is allowed for one-dimensional arrays only. The resulting array does not contain sorted array element values, but indices of the sorted values in the original array, which would otherwise be lost during sorting.

The following construct returns a *dense* array T, with $\mathcal{S}_T = [n]$ and $\mathcal{T}_T = \texttt{int}$:

```
T := topN(A, n, <ASC|DESC>)
```

The actual values can subsequently be fetched by element extraction from the original array, using indices stored in array T:

```
[ A(T(x)) | x ]
```

The domain of axis variable `x` can be implicit, as it is used to index array T, whose shape is known as $[n]$. Examples of these two steps are shown in Figure 4.2, at lines 22 and 25 respectively. Retrieval of the top-N values from multi-dimensional arrays (e.g. retrieve the 20 top values of each column of a matrix separately) is, as for sorting, not possible in standard relational algebra (see [CCG00] for a more complete analysis) and therefore not allowed in SRAM. Notice that, unlike for the sorting construct, the grouping cannot be expressed by means of sorting as a work-around.

Array concatenation

Array concatenation between two arrays appends the second array to the first one, along their first dimension. The shapes of the two arrays must therefore be identical, except for the first dimension, and their base types must be the same. Array concatenation is performed by the operator `++`:

```
C = A ++ B
```

with $|\mathcal{S}_A| = |\mathcal{S}_B|, \quad \mathcal{T}_A = \mathcal{T}_B, \quad \forall i \in]0, n[: \mathcal{S}_A(i) = \mathcal{S}_B(i)$.

For example, concatenating matrices $A[5, 10]$ and $B[3, 10]$ results in a matrix $C[8, 10]$: rows of B are appended to rows of A.

Concatenation along different dimensions is possible by pivoting arrays before and after concatenation. For example, concatenating columns of matrices $A[20, 7]$ and $B[20, 8]$, results in a matrix $C[20, 15]$:

```
C = mxTransp(mxTransp(A) ++ mxTransp(B))
```

where macro `mxTransp()` performs matrix transposition (see Section 4.3.8).

If-Then-Else

Conditional expressions can be specified using the `if-then-else` construct:

```
if (<boolean condition>) then <value a> else <value b>
```

where the type of `<value a>` and `<value b>` must be the same. This construct covers several needs, for example:

- selection of values from two different arrays, based on a condition

```
[ if(x<10) then A(x) else B(x) | x ]
```

- value-based selection, where, for example, $\varepsilon = 0$ could indicate absence of an element in the result array (this is an arbitrary choice made at application-level)

```
[ if(log(A(x)) > 2.0) then A(x) else 0.0 | x ]
```

- construction of arrays with specific value patterns, e.g. a diagonal matrix

```
[ if(x == y) then 1 else 0 | x<10,y<10 ]
```

4.3.7 Array reduction

Array reduction operations reduce an array to an atomic value. The two available operations of this type, array element extraction and array aggregation, are discussed hereafter.

Array element extraction

Array element extraction can be either an array construction operation, as described in Section 4.3.6, or an array reduction operation. The latter is true whenever element extraction is applied to a flat array, which yields an atomic value.

An example is shown in Figure 4.2, at Line 25. The expression `[ Cities(BestIdx(x)) | x ]` contains two array reductions by element extraction: `BestIdx(x)` returns an `int` value, while `Cities(BestIdx(x))` returns a `str` value.

Array aggregation

Array aggregation operators are defined as functions that entirely collapse any given n-dimensional array by aggregating all its values into a single atomic value. Possible aggregation functions include `sum`, `prod`, `avg`, `min`, `max`.

The following examples clarify how this simple definition allows for any complex aggregation pattern when used in combination with comprehension syntax.

Example 4.5 (Complete aggregation).
Given the 3-dimensional array $A_{x,y,z}$, the simplest case aggregates all its values into a single atomic value:

$$s = \sum_x \sum_y \sum_z A_{x,y,z}$$

The expression

```
s = sum(A)
```

or, explicitly in comprehension syntax

```
s = sum([ A(x,y,z) | x,y,z ])
```

returns the summation of all the elements in A, as an atomic value `s`. □

Example 4.6 (Partial aggregation).
Given the 3-dimensional array $A_{x,y,z}$, we want to sum over the `x` and `z` axes only:

$$B_y = \sum_x \sum_z A_{x,y,z}$$

This is achieved by exploiting the possibility of defining nested arrays.

In a hypothetical 2-step process, the first step is to reshape array A from its flat form

```
[ A(x,y,z) | x,y,z ]
```

into a nested variant, a vector (`y`) of matrices (`x,z`):

```
[ [ A(x,y,z) | x,z ] | y ]
```

For every instantiation $\texttt{y}_j \in [0, |\texttt{y}|[$ of the `y` axis, `[ A(x,y`$_j$`,z) | x,z ]` is a properly defined flat array with axes `x` and `z`. As a final step, the summation can be applied to this sub-array, which collapses axes `x` and `z` and yields the desired B_y vector:

```
B := [ sum([ A(x,y,z) | x,z ]) | y ]
```
□

4.3.8 Macro definition

Macros allow to write reusable expressions in SRAM scripts. Their syntax follows the pattern:

```
<macro name>(x,y,z,...) = <macro body>(x,y,z,...)
```

During a pre-processing phase, each macro-call in the script that matches `<macro name>` is replaced by the corresponding `<macro body>` definition.

If a macro has parameters (“`x,y,z,...`” in the syntax above), they are substituted into the macro body. Because no type check is performed on the macro definition itself, but only on the expanded body after the pre-processing phase, macro parameters can represent anything that is allowed in the syntax: atomic values, constants, arrays, and even other macro names, provided that those were previously defined. If no parameters are defined, the macro expands to a constant expression. Thus, a macro can mimic the definition of functions or constants, not otherwise available in the language. However, the macro mechanism does not provide useful features of functions such as dynamic type checking, argument stack management, recursivity. In-line functions are more similar to macros in that they perform the same type of body expansion, but dynamic type checking still differentiates the two constructs. A closer comparison can be made with the functionality provided by the `#define` directive of C/C++ pre-processors.

Figure 4.3 shows two examples of macro definitions. At Line 6 a non-parametric macro is used to define a constant. Line 10 shows a macro with two parameters, called `Temp` and `RainCm`. Because macro parameters are basically place-holders that can in principle represent any syntactic element, the fact that `Temp` and `RainCm` start with capital letters is simply a convention to indicate that the body of the corresponding macro is known to expect arrays in place of those parameters. This is particularly useful when the macro name gives no indication on the type of expected parameters.

It is good practice to organise commonly used macro definitions in *libraries*, i.e. scripts to be included with `#include` statements where needed. For example, a library script that defines macros for standard linear algebra operations would include the following code snippet:

```
// Matrix transposition. A is expected to be a matrix
mxTrnsp(A) = [ A(j,i) | i,j ]

// Matrix diagonal. A is expected to be a square matrix
mxDiag(A) = [ A(i,i) | i ]

// Matrix multiplication
// A and B are expected to be multiplication-compatible matrices
mxMult(A,B) = [ sum([ A(i,k) * B(k,j) | k ]) | i,j ]
```

4.4 Mapping **SRAM** syntax onto core array algebra

Comprehension syntax is a natural choice for describing the instantiation of arrays defined mathematically in Section 4.2.2. An array A is a function from its index domain $\mathcal{D}_A$ to its element type $\mathcal{T}_A$:

$$A : \mathcal{D}_A \mapsto \mathcal{T}_A$$

Comprehension syntax allows to define such a function explicitly, by specifying arrays in terms of their indices and elements, rather than as entire entities:

$$\texttt{A = [} f(\bar{\imath}) \texttt{ | } \bar{\imath} \in \mathcal{D}_A \texttt{]}$$

The direct advantage of this element-by-element approach at the user-interface level is that complex expressions can be described rather easily and the exact content of array elements can be derived precisely from their definition. When such a fine granularity is not desired, the macro mechanism can hide unnecessary details. For example, although SRAM syntax does not provide any built-in matrix multiplication operation, which therefore needs to be defined explicitly in comprehension syntax, its details can be hidden by a `mxMult()` macro:

```
Macro definitions
mxMult(A,B) = [ sum([ A(i,k) * B(k,j) | k ]) | i,j ]
mxTrnsp(A)  = [ A(j,i) | i,j ]

Macro calls
C = mxMult(A,B)
D = mxMult(C, mxTrnsp(C))
```

When the aim is to translate array expressions into a bulk-oriented language such as relational algebra, a representation based on comprehension syntax is not an ideal starting point. Having an intermediate translation step into an *array algebra* provides a twofold benefit: firstly, it reduces the semantic gap between array syntax and relational algebra, by operating on the array domain, as the former, but being a bulk-oriented algebra, as the latter; secondly, it facilitates tree rewriting-based optimisations that are specific of the array domain.

Table 4.2 briefly describes the main operators of this array algebra, also called *core array algebra* throughout this thesis. This core array algebra is extended with new operators in Chapter 5, for optimisation purposes.

Table 4.2: Core array operators

Array operator	Operation
$\mathit{Variable}(A_{\text{def}})$	binds an array declaration to the description A_{def} of its physical representation.
$\mathit{Grid}(\mathcal{S}, j)$	creates an array of shape $\mathcal{S}$, whose element values are the index values of its j-th dimension.
$\mathit{Const}(\mathcal{S}, c)$	creates an array of shape $\mathcal{S}$, whose element values are all equal to the given constant c.
$\mathit{Apply}(A, I_0, \ldots, I_{n-1})$	applies the n-dimensional array A, to n arrays whose element values are index values for A.
$\mathit{Map}(f, A_0, \ldots, A_{m-1})$	maps a m-ary function f to corresponding elements of shape-aligned arrays $A_0, \ldots, A_{m-1}$.
$\mathit{Aggregate}(f, j, A)$	collapses the last j dimensions by applying the aggregation function f.
$\mathit{Concat}(A, B)$	concatenates arrays A and B along their first dimension.
ISort(A, `<ASC|DESC>`)	for vectors only, it returns the indices of A's values in the desired order.
ITopN(A, n, `<ASC|DESC>`)	for vectors only, it returns the indices of the top n values of A in the desired order.

4.4.1 Normalisation of SRAM syntax

Before the actual translation of array expressions into array algebra, two normalisation steps, *flattening* and *shape alignment*, rewrite SRAM syntax expressions into a form that facilitates the recognition of specific patterns.

We present here motivations for and an informal description of the flattening and shape alignment normalisations. Formal translation rules to carry out this process are shown in Appendix A.1.

Flattening nested arrays

The array algebra operators are defined over *non-nested* arrays only. For this reason, a normalisation step called *flattening* converts nested arrays to their equivalent non-nested representations. For example, a nested array with type $([10, 5], ([20], \texttt{int}))$ is equivalent to a higher dimensionality, flat, array with type $([10, 5, 20], \texttt{int})$. A crucial rule of this normalisation step flattens a nested comprehension construct by appending the domain specification of the inner comprehension to the one of the outer comprehension. This means that the axes of the outer comprehension become the major axes in the new comprehension. For example:

```
[ [ f(x,y,z) | z<20 ] | x<10,y<5 ]
                 ↓
   [ f(x,y,z) | x<10,y<5,z<20 ]
```

Shape alignment

A second normalisation step reformulates array expressions to make them easier to handle for rule-based translation engines, by emphasising two aspects:

- element-by-element formulation of array expressions is a syntactic aid for writing expressions, but operations are defined over arrays to benefit bulk processing;
- all operations defined in array expressions are functions of the *result* array's index domain.

This step is called *shape alignment* in that it has the effect of reshaping every element in the expression to the result array's index domain. For example, consider the following expression and its equivalent after the shape alignment normalisation:

```
                    [ 2 + j | i<3,j<4 ]
                             ↓
[ [ 2 | i<3,j<4 ](i,j) + [ j | i<3,j<4 ](i,j) | i<3,j<4 ]
```

The result's index domain, identified by `[ | i<3,j<4 ]`, is propagated recursively to every element of the original expression (notice how `2` and `j` are expanded into an explicit array comprehension), making it become more clearly an expression involving shape-aligned arrays.

Rewriting SRAM expressions in terms of their result index domain allows for simple translation patterns to array algebra, because of the shape alignment effect.

Comparison to set-comprehension

It is important to emphasise that, although the shape-alignment procedure *has the effect* of reshaping every element in the original expression to the result's shape, its actual role is more strictly related to the formal definition of arrays:

the shape alignment procedure replicates the result's index domain for each element in the array expression and applies such an element to the indices created. This subtle difference between modifying an array (which could be the imperative programmer's solution) and creating a new array (the latter, functional programming approach being our goal) is best illustrated by the following example.

Example 4.7 (Shape alignment).
Selecting elements of a matrix in transposed order is not the same as transposing a matrix. Consider the following matrix A

$$A[3,4] = \begin{bmatrix} 1.0 & 1.5 & 3.0 & 3.0 \\ 1.5 & 1.0 & 1.5 & 3.0 \\ 3.0 & 1.5 & 1.0 & 1.5 \end{bmatrix}$$

and the following SRAM expression:

```
A' = [ A(y,x) | x<4,y<3 ]
```
(4.2)

We now break down the translation process of the expression in (4.2). As a first step, the shape of the result array A' is computed:

$$\mathcal{S}_{A'} = [4, 3].$$

The new array A' has two axes, named `x` and `y` in the expression. The shape alignment normalisation rewrites them in terms of index arrays, which we assign temporary names X and Y for clarity of presentation, with shape $[4, 3]$, as the result shape, containing their own x or y axis values respectively.

```
X = [ x | x<4,y<3 ]
Y = [ y | x<4,y<3 ]
```

Finally, expression `A(y,x)` translates to the application of array A to index arrays Y and X:

```
[ A(Y(x,y), X(x,y)) | x<4,y<3 ]
```

Putting it all together, the original SRAM expression translates as follows (for completeness, the array algebra intermediate translation is also shown, although only introduced in Section 4.5.3):

```
A' = [ A(y,x) | x<4,y<3 ]
```

$$\downarrow$$

```
[ A([ y | x<4,y<3 ](x,y), [ x | x<4,y<3 ](x,y)) | x<4,y<3 ]
```

$$\Downarrow \boxed{\mathsf{A}}$$

$$\mathit{Apply}(A, \mathit{Grid}([4,3],1), \mathit{Grid}([4,3],0))$$

$$\downarrow$$

$$A\left(\begin{bmatrix}0&1&2\\0&1&2\\0&1&2\\0&1&2\end{bmatrix},\begin{bmatrix}0&0&0\\1&1&1\\2&2&2\\3&3&3\end{bmatrix}\right)$$

$$\downarrow$$

$$\begin{bmatrix}A(0,0)&A(1,0)&A(2,0)\\A(0,1)&A(1,1)&A(2,1)\\A(0,2)&A(1,2)&A(2,2)\\A(0,3)&A(1,3)&A(2,3)\end{bmatrix}=\begin{bmatrix}1.0&1.5&3.0\\1.5&1.0&1.5\\3.0&1.5&1.0\\3.0&3.0&1.5\end{bmatrix}$$

□

We can easily recognise the definition of a matrix transposition in (4.2). However, it is more correct to describe that expression as the definition of a new matrix A', whose values are obtained by the application of matrix A (recall that arrays are functions) to indices y, x. The important distinction is that matrix A' is not a direct manipulation of matrix A, but rather a new matrix constructed by copying the values of A into A' in a different (transposed) order. The comprehension-style array constructor is called *generative*, because it always generates a new array and subsequently assigns each element a value, which is obtained by applying a function to the newly introduced indices. In the example, the function applied is array A itself.

For comparison, the same operation specified in a *set-oriented* syntax such as relational calculus could look like the following:

$$A' = \{\langle y, x, A(y,x)\rangle | x, y \in \mathcal{D}_{A_{x,y}}\} \tag{4.3}$$

This expression specifies that A' is a *set* of elements, here defined as $\langle y, x, A(y,x)\rangle$ tuples, where axes x and y belong to the index domain of array A.

The main difference between the set-oriented and the generative formulations of the matrix transposition is that whereas in (4.3) indices x and y are defined as belonging to A, they belong to A' in (4.2) and they have no formal relationship with A. Generalising, the indices used in comprehension syntax belong to the result array, whereas they belong to the input arrays in the set-oriented syntax. More formally, in the generative array comprehension-based approach, the value of each element of a new array A' is computed by a function $f : \mathcal{D}_{A'} \to \mathcal{T}_{A'}$. Note that the domain of such a function is $\mathcal{D}_{A'}$, hence the value of each element is a function of the *result* array's index values.

4.4.2 Mapping rules

This section defines the mapping rules from *normalised* SRAM syntax constructs (see Section 4.4.1 for normalisation) to array algebra operators. Note that, for clarity of presentation, the examples shown in this section do not include the normalisation phase, stepping from user-level SRAM syntax to array algebra directly. Each mapping rule is presented as a rule of inference (see Section 4.2.1), where the conclusion shows the resulting array algebra operator and its properties ($\mathcal{S}$, $\mathcal{T}$, ε, δ, etc). Some of these properties can be computed exactly, some other need estimations, whose quality depends on the information available in the framework. For example, a correct estimation for ε and δ often requires information on the distribution of ε values in the input arrays, as well as specific knowledge on functions that may be used (for example in *Map*). When such additional information is not available, conservative choices are made (e.g. uniform distributions, no prior knowledge on functions used). Note however, that wrong estimates for ε and δ can only affect performance and not correctness. Extending the framework to exploit additional information in order to achieve better estimates for array properties is considered future work.

Variable

Persistently stored arrays are included in array-algebra expressions using the operator *Variable*, obtained by applying the following equivalence rule:

$$\frac{\texttt{A = }(\mathcal{S},\mathcal{T})\texttt{ <sparse(}\varepsilon,\delta\texttt{)|dense> "<table name>"}}{\begin{array}{c}\texttt{A} \mapsto A = \mathit{Variable}(A_{\text{def}}) \\ A_{\text{def}} = \begin{cases} \mathcal{S}_A = \mathcal{S}, \quad \mathcal{T}_A = \mathcal{T} \\ \varepsilon_A = \varepsilon, \quad \delta_A = \delta \\ \text{storage}_A = \texttt{<table name>} \end{cases}\end{array}}$$

The *Variable* operator encapsulates all the information available from the data dictionary about the array associated to it: properties, physical storage, statistics, etc.

Example 4.8.

```
A = ([2,3],dbl) sparse(0,0.1) "Table_A"
```

$$\Big\Downarrow \boxed{\text{A}}$$

$$A = \mathit{Variable}(A_{\text{def}}), \qquad A_{\text{def}} = \begin{cases} \mathcal{S} = [2,3], \quad \mathfrak{T} = \texttt{dbl} \\ \varepsilon = 0, \quad \delta = 0.1 \\ \text{storage}_A = \texttt{"Table\_A"} \end{cases}$$

□

Grid

Indices introduced in comprehension expressions are translated to index arrays. The array-algebra operator that is used to represent index arrays is $\mathit{Grid}(\mathcal{S}, j)$:

$$\frac{\texttt{I = [i}_j\texttt{ | }\bar{\imath}\texttt{<}\mathcal{S}\texttt{]} \qquad \texttt{i}_j \in \bar{\imath}}{\begin{array}{c} \texttt{I} \mapsto I = \mathit{Grid}(\mathcal{S}, j) \\ \mathcal{S}_I = \mathcal{S} \qquad \mathcal{T}_I = \texttt{int} \\ \nexists \varepsilon_I \qquad \delta_I = 1 \end{array}}$$

The operator *Grid* creates an array of shape $\mathcal{S}$ and assigns its elements the value of its own indices in the j-th dimension, replicated in all the remaining dimensions. Any array produced by *Grid* is dense by definition. A similar operator is defined in [SS92].

Example 4.9.

```
[ x | x<2,y<3 ]                [ y | x<2,y<3 ]
```

$$\Big\Downarrow \boxed{\text{A}} \qquad\qquad\qquad \Big\Downarrow \boxed{\text{A}}$$

$$\mathit{Grid}([2,3],0) \qquad\qquad \mathit{Grid}([2,3],1)$$

$$\downarrow \qquad\qquad\qquad\qquad \downarrow$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \qquad\qquad \begin{bmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \end{bmatrix}$$

□

Const

Constant expressions result in arrays of the given shape, where each element is assigned a given constant c:

$$\frac{\texttt{A = [} c \texttt{ | } \bar{\imath}\texttt{<}\mathcal{S} \texttt{]}}{\begin{array}{c} \texttt{A} \mapsto A = \mathit{Const}(\mathcal{S}, c) \\ \mathcal{S}_A = \mathcal{S} \qquad \mathcal{T}_A = (\text{type of } c) \\ \varepsilon_A = c \qquad \delta_A = 0 \end{array}}$$

A constant array is by definition sparse, with $\varepsilon = c$ and $\delta = 0$.

Example 4.10.

$$\begin{tikzcd} \texttt{[1.0 | x<2,y<3]} \arrow[d, "\boxed{\text{A}}"] \\ \mathit{Const}([2,3], 1.0) \arrow[d] \\ \begin{bmatrix} 1.0 & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 \end{bmatrix} \end{tikzcd}$$

□

Apply

Index-based array operations such as element extraction, replication, pivoting and slicing, can be performed by the *Apply* operator:

$$\frac{\begin{array}{c} \texttt{A} \mapsto A \qquad \forall j \in [0, n[: \texttt{I}_j \mapsto I_j \\ \texttt{A' = [A(I}_0\texttt{(}\bar{\imath}\texttt{),...,I}_{n-1}(\bar{\imath})\texttt{) | } \bar{\imath}\texttt{<}\mathcal{S} \texttt{]} \end{array}}{\begin{array}{c} \texttt{A'} \mapsto A' = \mathit{Apply}(A, I_0, \ldots, I_{n-1}) \\ \mathcal{S}_{A'} = \mathcal{S} = \mathcal{S}_{I_0} = \cdots = \mathcal{S}_{I_{n-1}} \\ \mathcal{T}A' = \mathcal{T}_A \\ \varepsilon_{A'} = \varepsilon_A \qquad \delta_{A'} = \delta_A \end{array}}$$

Each index array I_j contains the j-th dimension coordinates of the values to select in A. Index arrays are limited by the following constraints: they must be dense and their base type must be `int`. Typically, such index arrays are generated by means of *Grid* operators.

The result of an *Apply* operator is an array whose shape is the same as that of index arrays $I_0, \ldots, I_{n-1}$, and whose type $\mathcal{T}_{A'}$ is the same as type $\mathcal{T}_A$ of array A. The set of unique element values that can possibly appear in A' is of course a subset of the ones appearing in A.

Example 4.11 (Creating a matrix with values taken from a vector).

$$A[4] = \begin{bmatrix} 1.0 \\ 1.5 \\ 2.0 \\ 2.5 \\ 3.0 \end{bmatrix}, \qquad I[4,4] = \begin{bmatrix} 0 & 1 & 4 & 4 \\ 1 & 0 & 1 & 4 \\ 4 & 1 & 0 & 1 \\ 4 & 4 & 1 & 0 \end{bmatrix}$$

```
A' = [ A(I(x,y)) | x<4,y<4 ]
```

$$\Big\Downarrow \boxed{\text{A}}$$

$$A' = Apply(A, I)$$

$$\downarrow$$

$$A'[4,4] = \begin{bmatrix} A(0) & A(1) & A(4) & A(4) \\ A(1) & A(0) & A(1) & A(4) \\ A(4) & A(1) & A(0) & A(1) \\ A(4) & A(4) & A(1) & A(0) \end{bmatrix} = \begin{bmatrix} 1.0 & 1.5 & 3.0 & 3.0 \\ 1.5 & 1.0 & 1.5 & 3.0 \\ 3.0 & 1.5 & 1.0 & 1.5 \\ 3.0 & 3.0 & 1.5 & 1.0 \end{bmatrix} \qquad \square$$

Example 4.12 (Taking a matrix diagonal).

$$A[4,4] = \begin{bmatrix} 1.0 & 1.5 & 3.0 & 3.0 \\ 1.5 & 1.0 & 1.5 & 3.0 \\ 3.0 & 1.5 & 1.0 & 1.5 \\ 3.0 & 3.0 & 1.5 & 1.0 \end{bmatrix}, \qquad I_0[4] = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \end{bmatrix}, \qquad I_1[4] = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \end{bmatrix}$$

```
A' = [ A(I0(x),I1(x)) | x<4 ]
```

$$\Big\Downarrow \boxed{\text{A}}$$

$$A' = Apply(A, I_0, I_1)$$

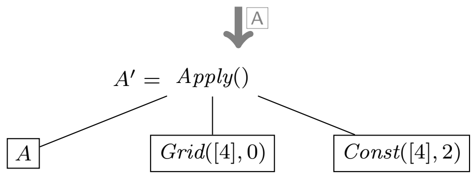
$$\downarrow$$

$$A'[4] = \begin{bmatrix} A(0,0) \\ A(1,1) \\ A(2,2) \\ A(3,3) \end{bmatrix} = \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{bmatrix} \qquad \square$$

Example 4.13 (Slicing a matrix to obtain a column).

$$A[4,4] = \begin{bmatrix} 1.0 & 1.5 & 3.0 & 3.0 \\ 1.5 & 1.0 & 1.5 & 3.0 \\ 3.0 & 1.5 & 1.0 & 1.5 \\ 3.0 & 3.0 & 1.5 & 1.0 \end{bmatrix}$$

```
A’ = [ A(x,2) | x<4 ]
```



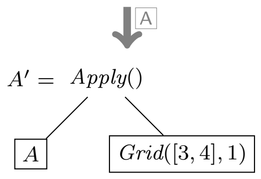
$$\downarrow$$

$$A'[4] = \begin{bmatrix} A(0,2) \\ A(1,2) \\ A(2,2) \\ A(3,2) \end{bmatrix} = \begin{bmatrix} 3.0 \\ 1.5 \\ 1.0 \\ 1.5 \end{bmatrix} \qquad \square$$

Example 4.14 (Replicating a vector up to a matrix).

$$A[4] = \begin{bmatrix} 3.0 \\ 1.5 \\ 1.0 \\ 1.5 \end{bmatrix}$$

```
A’ = [ A(y) | x<3, y<4 ]
```



$$\downarrow$$

$$A'[3,4] = \begin{bmatrix} A(0) & A(1) & A(2) & A(3) \\ A(0) & A(1) & A(2) & A(3) \\ A(0) & A(1) & A(2) & A(3) \end{bmatrix} = \begin{bmatrix} 3.0 & 1.5 & 1.0 & 1.5 \\ 3.0 & 1.5 & 1.0 & 1.5 \\ 3.0 & 1.5 & 1.0 & 1.5 \end{bmatrix} \qquad \square$$

Map

The operator *Map* maps an m-ary function f to corresponding elements of arrays $A_0, \dots, A_{m-1}$:

$$\frac{\begin{array}{c} \forall l \in [0, m[\colon \texttt{A}_l \mapsto A_l \\ f : \mathcal{T}_{A_0} \times \cdots \times \mathcal{T}_{A_{m-1}} \to \mathcal{T}_f \\ \texttt{A' = [f(A}_0\texttt{(}\bar{\imath}\texttt{), ..., A}_{m-1}\texttt{(}\bar{\imath}\texttt{)) | }\bar{\imath}\texttt{<}\mathcal{S}\texttt{]} \end{array}}{\begin{array}{c} \texttt{A'} \mapsto A' = \mathit{Map}(f, A_0, \dots, A_{m-1}) \\ \mathcal{S}_{A'} = \mathcal{S} \qquad \mathcal{T}_{A'} = \mathcal{T}_f \\ \varepsilon_{A'} = f(\varepsilon_{A_0}, \dots, \varepsilon_{A_{m-1}}) \qquad \delta_{A'} = \mathit{max}(\delta_{A_0}, \dots, \delta_{A_{m-1}}) \end{array}}$$

Estimation of the sparsity properties of the output array takes a conservative approach, by assuming a large index overlap among the default elements of the arrays in input.

Example 4.15 (Taking the log *of every value in a matrix).*

$$A = \begin{bmatrix} 1.0 & 1.5 & 3.0 & 3.0 \\ 1.5 & 1.0 & 1.5 & 3.0 \\ 3.0 & 1.5 & 1.0 & 1.5 \\ 3.0 & 3.0 & 1.5 & 1.0 \end{bmatrix}$$

```
A' = [ log(A(x,y)) | x<4,y<4 ]
```

$$\Big\Downarrow \boxed{\text{A}}$$

$$A' = \mathit{Map}(\log, A)$$

$$\downarrow$$

$$A' = \begin{bmatrix} \log(1.0) & \log(1.5) & \log(3.0) & \log(3.0) \\ \log(1.5) & \log(1.0) & \log(1.5) & \log(3.0) \\ \log(3.0) & \log(1.5) & \log(1.0) & \log(1.5) \\ \log(3.0) & \log(3.0) & \log(1.5) & \log(1.0) \end{bmatrix} \qquad \square$$

Example 4.16 (Value-by-value addition).

$$A = \begin{bmatrix} 1.0 & 1.5 & 3.0 & 3.0 \\ 1.5 & 1.0 & 1.5 & 3.0 \\ 3.0 & 1.5 & 1.0 & 1.5 \\ 3.0 & 3.0 & 1.5 & 1.0 \end{bmatrix}, \qquad B = \begin{bmatrix} 0.0 & 1.5 & 1.0 & 2.0 \\ 0.5 & 1.0 & 1.5 & 2.0 \\ 0.0 & 1.5 & 1.5 & 1.0 \\ 0.0 & 1.0 & 1.5 & 2.0 \end{bmatrix}$$

```
C = [ A(x,y) + B(x,y) | x<4,y<4 ]
```

$$\Big\Downarrow \boxed{\text{A}}$$

$$C = \mathit{Map}(+, A, B)$$

$$\downarrow$$

$$C = \begin{bmatrix} A(0,0)+B(0,0) & A(0,1)+B(0,1) & A(0,2)+B(0,2) & A(0,3)+B(0,3) \\ A(1,0)+B(1,0) & A(1,1)+B(1,1) & A(1,2)+B(1,2) & A(1,3)+B(1,3) \\ A(2,0)+B(2,0) & A(2,1)+B(2,1) & A(2,2)+B(2,2) & A(2,3)+B(2,3) \\ A(3,0)+B(3,0) & A(3,1)+B(3,1) & A(3,2)+B(3,2) & A(3,3)+B(3,3) \end{bmatrix}$$
$$= \begin{bmatrix} 1.0 & 3.0 & 4.0 & 5.0 \\ 2.0 & 2.0 & 3.0 & 5.0 \\ 3.0 & 3.0 & 2.5 & 2.5 \\ 3.0 & 4.0 & 3.0 & 3.0 \end{bmatrix}$$

□

The `if-then-else` construct

The `if-then-else` construct of the SRAM syntax needs no dedicated array algebra operator, as it can be translates to the *Map* of a function *ifthenelse*:

$$\frac{\begin{array}{c} \texttt{C} \mapsto C \qquad \texttt{A} \mapsto A \qquad \texttt{B} \mapsto B \\ \mathcal{T}_C = \texttt{bit} \qquad \mathcal{T}_A = \mathcal{T}_B \\ \texttt{A' = [if(C(}\bar{\imath}\texttt{)) then A(}\bar{\imath}\texttt{) else B(}\bar{\imath}\texttt{)) | }\bar{\imath}\texttt{<}\mathcal{S}\texttt{]} \end{array}}{\begin{array}{c} \texttt{A'} \mapsto A' = \mathit{Map}(\mathit{ifthenelse}, C, A, B) \\ \mathcal{S}_{A'} = \mathcal{S} \qquad \mathcal{T}_{A'} = \mathcal{T}_A = \mathcal{T}_B \\ \varepsilon_{A'} = \varepsilon_B \qquad \delta_{A'} = \delta_B \end{array}}$$

Predicting $\varepsilon_{A'}$ and $\delta_{A'}$ is very hard without additional knowledge on the arrays in input and on the condition that yields boolean array C. The arbitrary choice of assigning default value and density of array B gives the possibility of steering a more efficient translation at SRAM syntax level, where some of this knowledge may be available. Given that choice, one would write the boolean condition in such a way that the `else` branch, that corresponds to array B, is the one that is selected most frequently.

Aggregate

Array aggregation is supported by the operator *Aggregate*, with the following syntax:

$$\frac{\begin{array}{c}\texttt{A} \mapsto A \\ f : \{\mathcal{T}_A\} \to \mathcal{T}_f \\ \texttt{A' = aggr(f, j, A)}\end{array}}{\begin{array}{c}\texttt{A'} \mapsto A' = \mathit{Aggregate}(f, j, A) \\ \mathcal{S}_{A'} = [\mathcal{S}_A(0), \ldots, \mathcal{S}_A(n-j-1)] \qquad \mathcal{T}_{A'} = \mathcal{T}_f \\ \nexists \varepsilon_{A'} \qquad \delta_{A'} = 1\end{array}}$$

Recall from Section 4.3.7 how nested array semantics is exploited in order to express partial aggregation. During the shape-alignment phase (see Section 4.4.1 and Appendix A.2), the j dimensions that are to be collapsed by the aggregation function are moved to the last j positions in the internal representation of array A in the algebra expression. The j parameter of the *Aggregate* operator represents precisely the number of dimensions that are dropped by the aggregation function f. This semantics simplifies the translation, without compromising the possibility of aggregating along any dimension, by first performing an axis re-ordering. Possible aggregation functions include *sum*, *prod*, *avg*, *min*, *max*. The result of an aggregation is defined as dense, also when the input array is sparse. This is a conservative choice that is due to the difficulty of estimating, in general, correct values for ε and δ. Investigating this issue is part of future research. Possible solutions include the management of detail statistics on the data distribution at hand, perhaps even provided by the user directly, or run-time estimations based on a sample of data.

Concat

The operator *Concat* concatenates two arrays, by appending the second to the first one. The concatenation happens, by an arbitrary design choice, along the first dimension. For examples, given two matrices A and B with equal number of columns, the *Concat* operator appends B's rows to A's rows:

$$\frac{\begin{array}{c}\texttt{A} \mapsto A \qquad \texttt{B} \mapsto B \\ \mathcal{T}_A = \mathcal{T}_B \qquad |\mathcal{S}_A| = |\mathcal{S}_B| = n \qquad \forall i \in]0, n[: \mathcal{S}_A(i) = \mathcal{S}_B(i) \\ \texttt{C = A ++ B}\end{array}}{\begin{array}{c}\texttt{C} \mapsto C = \mathit{Concat}(A, B) \\ \mathcal{S}_C = [\mathcal{S}_A(0) + \mathcal{S}_B(0), \mathcal{S}_A(1), \ldots, \mathcal{S}_A(n-1)] \qquad \mathcal{T}_C = \mathcal{T}_A \\ \varepsilon_C = \begin{cases} \varepsilon_A & \text{if } \varepsilon_A = \varepsilon_B \\ \varepsilon_A & \text{if } \varepsilon_A \neq \varepsilon_B \wedge |A_\varepsilon| \geq |B_\varepsilon| \\ \varepsilon_B & \text{if } \varepsilon_A \neq \varepsilon_B \wedge |A_\varepsilon| < |B_\varepsilon| \end{cases} \\ \delta_C = \begin{cases} \dfrac{|A|\delta_A + |B|\delta_B}{|A| + |B|} & \text{if } \varepsilon_A = \varepsilon_B \\ \delta_A & \text{if } \varepsilon_A \neq \varepsilon_B \wedge |A_\varepsilon| \geq |B_\varepsilon| \\ \delta_B & \text{if } \varepsilon_A \neq \varepsilon_B \wedge |A_\varepsilon| < |B_\varepsilon| \end{cases} \\ \text{where } |X_\varepsilon| = |X|(1 - \delta_X)\end{array}}$$

When the two arrays in input have the same default value ε, this is also the default value of the result array, with density δ trivially recomputed. When $\varepsilon_A \neq \varepsilon_B$, the default value and density of the array with the most non-stored values (see definition of $|X_\varepsilon|$) is chosen.

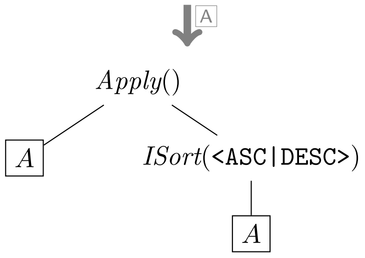
ISort

For vectors only, *ISort* returns the indices of an array's values in the desired order.

$$\frac{\begin{array}{c}\texttt{A} \mapsto A \\ |\mathcal{S}_A| = 1 \\ \texttt{S = sort(A,<ASC|DESC>)}\end{array}}{\begin{array}{c}\texttt{S} \mapsto S = \mathit{ISort}(A, \texttt{<ASC|DESC>}) \\ \mathcal{S}_S = \mathcal{S}_A \qquad \mathcal{T}_S = \texttt{int} \\ \nexists \varepsilon_S \qquad \delta_S = 1\end{array}}$$

The actual values can subsequently be fetched by element extraction from the original array, using indices stored in array S:

```
S = sort(A,<ASC|DESC>)
[ A(S(x)) | x ]
```



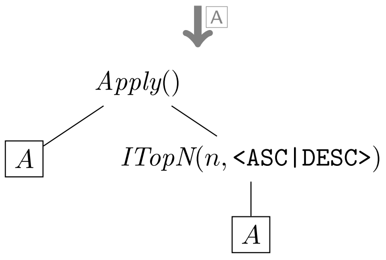
ITopN

For vectors only, *ITopN* returns the indices of the top n values in the desired order.

$$\frac{\texttt{A} \mapsto A \quad |\mathcal{S}_A| = 1 \quad \texttt{T = topN(A,n,<ASC|DESC>)}}{\texttt{T} \mapsto T = \mathit{ITopN}(A, n, \texttt{<ASC|DESC>}) \quad \mathcal{S}_T = [n] \quad \mathfrak{T}_T = \texttt{int} \quad \nexists \varepsilon_T \quad \delta_T = 1}$$

The actual values can subsequently be fetched by element extraction from the original array, using indices stored in array T:

```
T = topN(A,n,<ASC|DESC>)
[ A(T(x)) | x ]
```



4.5 Mapping array algebra onto relational algebra

This section explains the most important translation step in the framework, that maps expressions from the array domain onto the relational domain. This entails the definition of procedures aimed at addressing the following two aspects:

- How are arrays mapped onto relations?
- How are array operations mapped onto relational operations?

Rules for these mappings are shown in detail in sections 4.5.1 and 4.5.3. The same sections introduce some restrictions as well as some extensions that are applied to the relational algebra used throughout this thesis.

4.5.1 Relational representation of sparse arrays

Outside the scope of relational domain, many storage schemes have been proposed for storing sparse multi-dimensional arrays, with strong emphasis on the special case of matrices [DDRvdV00] and more in particular sparse matrices [DER86]. Most of them are tuned for specific data access patterns (e.g. Compressed Row/Column Storage, Compressed Diagonal Storage, Skyline Storage) that are required by specific applications.

A generic and simple choice for storing sparse n-dimensional arrays in relational databases maps every array to one relation, and every array element to a tuple in a relation. Table 4.3 summarises the notation used throughout this thesis for relations that represent sparse arrays. Notice that arrays and their associated relations are denoted by a different text font: an array A (see Table 4.1) is stored in the database as a relation A.

Rule 4.1 (Relational mapping of a sparse array). An n-dimensional sparse array A, with default value ε_A, is mapped onto a relation A, where each tuple corresponds to one array element and enumerates index values and element value explicitly. Only those elements for which $A(\bar{\imath}) \neq \varepsilon_A$ need to be represented in the relation:

$$\frac{\begin{array}{c} A_{\bar{\imath}} \\ \bar{\imath} = (i_0, \ldots, i_{n-1}) \end{array}}{\begin{array}{c} A \mapsto \mathsf{A}(\bar{\mathsf{\imath}}, \mathsf{v}) = \{(\bar{\imath}, A(\bar{\imath})) \mid \bar{\imath} \in \mathcal{D}_A, A(\bar{\imath}) \neq \varepsilon_A\} \\ \bar{\mathsf{\imath}} = \mathsf{i}_0, \ldots, \mathsf{i}_{n-1} \end{array}}$$

Index columns together form the primary key of relation A. □

Table 4.3: Relational algebra notation for arrays

Notation	Meaning
A	relation A corresponding to array A
$\lvert\mathsf{A}\rvert$	size (number of rows) of relation A
$\mathsf{A.a}$	generic attribute a of relation A
a	generic attribute a of the relation in context
$\mathsf{A.i}_j$	attribute i_j of relation A corresponding to the j-th index of array A
$\mathsf{A.v}_j$	attribute v_j of relation A corresponding to the j-th value of array A
$\mathsf{A.v}$	$\mathsf{A.v} \equiv \mathsf{A.v}_0$
$\mathsf{A.\bar{i}}$	attributes $\mathsf{A.i}_0, \ldots, \mathsf{A.i}_{n-1}$ corresponding to indices $i_0, \ldots, i_{n-1}$ of array A
$\mathsf{A.\bar{v}}$	attributes $\mathsf{A.v}_0, \ldots, \mathsf{A.v}_{m-1}$ corresponding to values $v_0, \ldots, v_{m-1}$ of array A
c	a constant value c
rowid()	identity column: a dense sequence of integers starting from 0

This representation scheme is particularly convenient for the following reasons:

- it implies a one-to-one relationship between arrays and relations;
- it implies a one-to-one relationship between array elements and tuples.
- it naturally includes dense arrays, for which no ε value is specified, and therefore all the tuples are represented;

Because the array algebra layer is independent of its physical (relational in this case) representation, other choices could be considered, provided that operations are mapped accordingly between the two domains. Section 5.2.1 lists some alternative storage schemes, that may provide more compact storage and better performance in some cases.

The relational scheme proposed makes explicit the possibility for arrays to contain more than one value per element (see Definition 4.4). Although SRAM syntax and array algebra are tuned for handling single-value arrays, multi-value arrays can be represented easily by extending the list of value attributes to $\mathsf{v}_0, \ldots, \mathsf{v}_{m-1}$, as shown in Table 4.1, which yields an array A-compatible schema $\mathsf{A}(\bar{\mathsf{i}}, \bar{\mathsf{v}})$. Relations that represent multi-value arrays are introduced to represent the intermediate results of some of the mapping rules in Section 4.5.3 (e.g. Rule 4.7). Note however that Rule 4.1 needs not be extended, as it deals with those single-value arrays that are visible explicitly at syntax and algebra level.

Table 4.4: Relational algebra operators

Operation	Name	Comments / restrictions
$\sigma_p(A)$	Selection	$p =$ boolean combination of $\{i_j \; \theta \; i_k, i_j \; \theta \; c\}$
		$\theta \in \{<, \leq, =, \neq, \geq, >\}$
$\pi_{a'_1,\ldots,a'_m}(A)$	Projection	$\forall a'_i \in \{a'_1, \ldots, a'_m\} : a'_i = f_i(A.a_1, \ldots, A.a_n)$,
${}_{\bar{\imath}'}\mathcal{G}_{v_0=g(v_0)}(A)$	Aggregation	$\bar{\imath}' \subset A.\bar{\imath}, \quad g \in \{sum, prod, min, max\}$
$A \cup B$	Union	
$A \setminus B$	Difference	
$A \times B$	Cartesian product	
$A \bowtie_p B$	Equi-join	$p =$ conjunctive combination of $\{A.a = B.b, a\}$
		where a means $A.a = B.a$
$A =\!\bowtie_p B$	Left outer-join	
$A =\!\bowtie\!=_p B$	Full outer-join	
$A \ltimes_p B$	Semi-join	$\pi_{A.\bar{\imath}, A.\bar{v}}(A \bowtie_p B)$
$A \rhd_p B$	Anti-join	$A \setminus (A \ltimes_p B)$
$A \overleftarrow{=\!\bowtie}_{\bar{\imath}} B$		$\pi_{A.\bar{\imath}, \bar{v}}(A =\!\bowtie_{\bar{\imath}} B)$
$A =\!\check{\bowtie}\!=_{\bar{\imath}} B$		$\pi_{(\forall i \in \bar{\imath}:\ A.i \vee B.i), \bar{v}}(A =\!\bowtie\!=_{\bar{\imath}} B)$
$\mathit{IndexTable}(\mathcal{S})$		populates a relation $G(i_0, \ldots, i_{\lvert\mathcal{S}\rvert-1})$
		with the enumeration of index values defined by $\mathcal{S}$
$\tilde{A}$		$\pi_{\bar{\imath}, v=\varepsilon_A}(\mathit{IndexTable}(\mathcal{S}_A) \setminus \pi_{\bar{\imath}}(A))$

4.5.2 Relational algebra operators

This section introduces the operators used in the relational algebra layer, summarised in Table 4.4. When compared to standard relational algebra, some predicates are restricted while a few extra operations are defined.

The projection operator (π) is supported as in standard relational algebra. It is mostly used to compute new array values, modify array indices, and compose new relations with array A-compatible schemas $A(\bar{\imath}, \bar{v})$.

The selection operator (σ_p) is only used to select tuples based on the value of index attribute, which corresponds to allowing index-based selections only in the array domain. Predicate p is a boolean combination of conditions on index attributes, that can use any of the following relations: $\{<, \leq, =, \neq, \geq, >\}$.

The aggregation operator ($\mathcal{G}$) works as in standard relational algebra, except that it is only used to reduce the number of index attributes and compute the aggregate value on the first value attribute of the input relation and to the first value of the output relation.

Union ($\cup$), difference ($\setminus$) and cartesian product ($\times$) operators are supported with no extensions nor restrictions with respect to standard relational algebra.

A number of different join operators are used. All of them indicate join conditions as a predicate p in subscript, that is a conjunctive combination of equality conditions. Each equality condition is expressed in one of the following forms:

- $\mathsf{A.a} = \mathsf{B.b}$
- $\mathsf{a} \quad \equiv \quad \mathsf{A.a} = \mathsf{B.a}$
  (A and B are the left and the right relations respectively)
- $\mathsf{A.\bar{\imath}} = \mathsf{B.\bar{\imath}} \quad \equiv \quad \forall j \in [0, L[\colon \mathsf{A.i}_j = \mathsf{B.i}_j, \quad L = |\mathsf{A.\bar{\imath}}| = |\mathsf{B.\bar{\imath}}|$
- $\bar{\imath} \quad \equiv \quad \mathsf{A.\bar{\imath}} = \mathsf{B.\bar{\imath}}$
  (A and B are the left and the right relations respectively)

The standard equi-join operator ($\bowtie_p$) is the only one used in this chapter. When presenting some optimisation techniques, in Chapter 5, the following join operators are also used: left outer-join ($=\!\bowtie_p$), full outer-join ($=\!\bowtie\!=_p$), semi-join ($\ltimes_p$) and anti-join ($\triangleright_p$).

Compound operators $\overleftarrow{=\!\bowtie}_{\bar{\imath}}$ and $=\!\check{\bowtie}\!=_{\bar{\imath}}$ perform a projection on top of left- and outer-join respectively, in order to select only the non-`NULL` attributes from the join outputs. They should be used only under the assumptions explained in Section 5.2.3.

is the relational implementation of array operator *Grid* (see Section 4.5.3).

Function $\mathit{IndexTable}(\mathcal{S})$ creates a relation $G(i_0, \ldots, i_{n-1})$ filled with the enumeration of all index values in the $\mathcal{D}$ domain defined by $\mathcal{S}$. This forms the basis for the relational mapping of the *Grid* operator, as shown in Rule 4.4 below.

Given the relational representation for sparse arrays defined in Section 4.5.1, the support for sparse array computations in terms of relational queries may require materialisation of the non-stored portion of a sparse array relation. Because the nominal $\mathcal{S}_A$ of array A is known, the domain of the index columns in the associated relational table A is known as well, which makes the non-stored portion of the relation, denoted as $\tilde{\mathsf{A}}$, always computable as:

$$\tilde{\mathsf{A}} = \pi_{\bar{\imath}, \mathsf{v}=\varepsilon_A}(\mathit{IndexTable}(\mathcal{S}_A) \setminus \pi_{\bar{\imath}}(\mathsf{A}))$$

4.5.3 Mapping rules

Given a tree structure representing an array algebra expression, the mapping rules presented in this section are applied in a bottom-up fashion, which yields a new tree structure representing the corresponding relational expression.

Array algebra operators may produce sparse or dense arrays in output, have dense or sparse arrays as inputs, or have no input array at all. In each case, the complexity of the array-to-relational mapping rules varies depending on the necessity of taking care of the non-stored portion of sparse arrays, both in the input relations and in the output relation. The mapping rules defined in this section yield correct results for both sparse and dense computations. In general, mappings for sparse array A include some extra processing needed for handling non-stored elements, denoted as $\tilde{\mathsf{A}}$ (Chapter 5 discusses optimisation techniques aimed at reducing this additional cost). By applying Rule 4.2, sub-expressions containing $\tilde{\mathsf{A}}$ are removed in presence of dense computations.

Rule 4.2 (Empty $\tilde{\mathsf{A}}$). When array A is dense, its default value ε_A is not defined. Its relational representation is fully stored, which makes $\tilde{\mathsf{A}}$ an empty relation:

$$\frac{A \mapsto \mathsf{A} \qquad \nexists \varepsilon_A}{\tilde{\mathsf{A}} \equiv \{\}} \qquad \square$$

Variable

Persistently stored arrays are included in array-algebra expressions using the *Variable* operator:

Rule 4.3 (Relational mapping for *Variable*).

$$\frac{A = \mathit{Variable}(A_{\text{def}})}{A \mapsto (A_{\text{def}}.\text{storage}) = \mathsf{A}} \qquad \square$$

Grid

The array operator *Grid* cannot be implemented by any standard relational operator. We assume an extended relational algebra that provides a $\mathsf{IndexTable}(\mathcal{S})$ operator, which creates a table with the enumeration of all indices defined by shape $\mathcal{S}$ (see Section 4.5.2 for further details).

Rule 4.4 (Relational mapping for *Grid*).

$$\frac{A = \mathit{Grid}(\mathcal{S}, j)}{A \mapsto \mathsf{A} = \pi_{\bar{\mathsf{I}}, \mathsf{v} = \mathsf{i}_j}(\mathsf{IndexTable}(\mathcal{S}))} \qquad \square$$

The projection on top of the $\mathsf{IndexTable}(\mathcal{S})$ relation constructs the index array, by propagating all index columns and replicating the j-th index columns as a value column.

Already in one of the first array query languages, AQL [LMW96], it is shown that a prerequisite for the implementation of array structures and computations in a relational system is the availability of a generator for intervals of integer numbers. Although no standard relational operator can generate such a relation, most modern RDBMS can easily be extended to provide such a functionality. Support for *table functions*, i.e. external functions that can return a table, is also included in the SQL-2003 standard [EMK$^+$04]. Note that this functionality cannot be provided by mechanisms such as the `RANK() OVER` window function (also defined in SQL-2003 standard), which can only add a column with integer identifiers based on the values of an existing table.

Const

A constant array can be mapped to a relational expression that projects the constant c on the value attribute of a table representing the index domain specified by a shape $\mathcal{S}$.

Rule 4.5 (Relational mapping for *Const*).

$$\frac{A = \mathit{Const}(\mathcal{S}, c)}{A \mapsto \mathsf{A} = \pi_{\bar{\imath}, v=c}(\mathsf{IndexTable}(\mathcal{S}))} \qquad \square$$

Recall that a constant array is fully sparse by definition, with $\varepsilon = c$ and $\delta = 1$. In practice, this means that the relational expression defined by Rule 4.5 never needs materialisation.

Apply

Selection of array element values based on their array index is performed by the *Apply* operator:

$$\mathit{Apply}(A, I_0, \ldots, I_{n-1})$$

Each index array I_j contains in its value column the j-th dimension coordinate of the values to select in A. Index arrays are dense and their base value must be of integer type. Rule 4.6 defines this operator by performing a series of primary key joins between relations $\mathsf{I}_0 \cdots \mathsf{I}_{n-1}$ – to construct a new relation $\mathsf{X}(\bar{\imath},\bar{v})$ that combines all value columns into one tuple per array index – and a foreign-primary key join ($\mathsf{X}.\bar{v} = \mathsf{A}.\bar{\imath}$) to retrieve the correct values from relation A.

Rule 4.6 (Relational mapping for *Apply*).

$$\frac{\begin{array}{c} A \mapsto \mathsf{A} \qquad \forall j \in [0,n[: I_j \mapsto \mathsf{I}_j \\ B = \mathit{Apply}(A, I_0, \ldots, I_{n-1}) \end{array}}{\begin{array}{c} B \mapsto \mathsf{B} = \pi_{\bar{\mathsf{i}}=\mathsf{X}.\bar{\mathsf{i}}, \mathsf{v}=\mathsf{A}.\mathsf{v}}\,(\mathsf{X} \bowtie_{\mathsf{X}.\bar{\mathsf{v}}=\mathsf{A}.\bar{\mathsf{i}}} \mathsf{A}) \\ \mathsf{X} = \mathsf{I}_0 \bowtie_{\bar{\mathsf{i}}} \cdots \bowtie_{\bar{\mathsf{i}}} \mathsf{I}_{n-1} \end{array}} \qquad \square$$

This rule remains identical when applied to sparse and dense arrays.

Map

The operator *Map* maps an m-ary function f to corresponding elements of arrays $A_0, \ldots, A_{m-1}$. This results in two relational steps:

1. (Join) perform a join on corresponding indices to gather input arrays' element values in one tuple per array index.

2. (Projection) apply function f to values gathered in step 1.

The simple case of a function applied to one array, $\mathit{Map}(f, A)$, needs not perform step 1, and translates as in (4.4) for both dense and sparse evaluation.

$$\frac{\begin{array}{c} A \mapsto \mathsf{A} \\ B = \mathit{Map}(f, A) \end{array}}{B \mapsto \mathsf{B} = \pi_{\bar{\mathsf{i}}, \mathsf{v}=f(\mathsf{A}.\mathsf{v})}(\mathsf{A})} \tag{4.4}$$

Consider now the case of two shape-aligned *dense* arrays A and B and the expression $\mathit{Map}(f, A, B)$. This maps to a join over index columns, followed by a projection on value columns:

$$\frac{\begin{array}{c} A \mapsto \mathsf{A} \qquad \nexists \varepsilon_A \\ B \mapsto \mathsf{B} \qquad \nexists \varepsilon_B \\ C = \mathit{Map}(f, A, B) \end{array}}{C \mapsto \mathsf{C} = \pi_{\bar{\mathsf{i}}, \mathsf{v}=f(\mathsf{A}.\mathsf{v}, \mathsf{B}.\mathsf{v})}(\mathsf{A} \bowtie_{\bar{\mathsf{i}}} \mathsf{B})} \tag{4.5}$$

If shape-aligned arrays A and B are *sparse*, with ε_A and ε_B, the array algebra expression $\mathit{Map}(f, A, B)$ requires a more complex relational mapping, in order to take care of the non-stored portion of arrays A and B, denoted as $\tilde{\mathsf{A}}$ and $\tilde{\mathsf{B}}$. Because sparse arrays consist of two parts `(stored,omitted)`, the join phase of a *Map* operation results in the union of four combinations, `(stored,omitted)`$\times$`(stored,omitted)`, and the *Map* consists of the union of four relational joins:

$$\frac{A \mapsto \mathsf{A} \qquad B \mapsto \mathsf{B} \qquad C = \mathit{Map}(f, A, B)}{C \mapsto \mathsf{C} = \bigcup_{j=1}^{4} \mathsf{C}_j} \tag{4.6}$$

$$\begin{aligned}
\mathsf{C}_1 &= \pi_{\check{\mathsf{I}},\mathsf{v}=f(\mathsf{A.v},\mathsf{B.v})}(\mathsf{A} \bowtie_{\bar{\mathsf{I}}} \mathsf{B})\\
\mathsf{C}_2 &= \pi_{\check{\mathsf{I}},\mathsf{v}=f(\mathsf{A.v},\varepsilon_B)}(\mathsf{A} \bowtie_{\bar{\mathsf{I}}} \tilde{\mathsf{B}})\\
\mathsf{C}_3 &= \pi_{\check{\mathsf{I}},\mathsf{v}=f(\varepsilon_A,\mathsf{B.v})}(\tilde{\mathsf{A}} \bowtie_{\bar{\mathsf{I}}} \mathsf{B})\\
\mathsf{C}_4 &= \pi_{\check{\mathsf{I}},\mathsf{v}=f(\varepsilon_A,\varepsilon_B)}(\tilde{\mathsf{A}} \bowtie_{\bar{\mathsf{I}}} \tilde{\mathsf{B}})
\end{aligned}$$

Notice how Rule 4.2 converts (4.6) into (4.5), should arrays A and B be dense. The rule removes expressions C_2, C_3 and C_4 from (4.6). The same rule removes either C_2 or C_3, together with C_4, in case of a combination of dense and sparse input arrays.

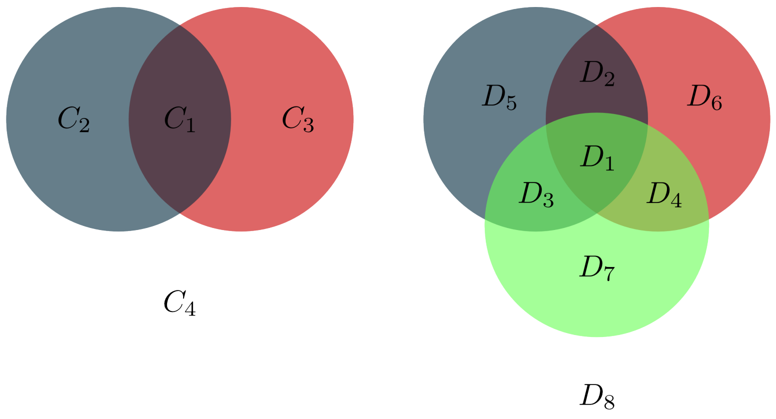
The translation rule for three sparse shape-aligned arrays A, B and C is similar. The join phase involves in this case the `(stored,omitted)` parts of three arrays, which yields $2^3 = 8$ combinations:

$$\frac{A \mapsto \mathsf{A} \qquad B \mapsto \mathsf{B} \qquad C \mapsto \mathsf{C} \qquad D = \mathit{Map}(f, A, B, C)}{D \mapsto \mathsf{D} = \bigcup_{j=1}^{8} \mathsf{D}_j} \tag{4.7}$$

$$\begin{aligned}
\mathsf{D}_1 &= \pi_{\check{\mathsf{I}},\mathsf{v}=f(\mathsf{A.v},\mathsf{B.v},\mathsf{C.v})}(\mathsf{A} \bowtie_{\bar{\mathsf{I}}} \mathsf{B} \bowtie_{\bar{\mathsf{I}}} \mathsf{C})\\
\mathsf{D}_2 &= \pi_{\check{\mathsf{I}},\mathsf{v}=f(\mathsf{A.v},\mathsf{B.v},\varepsilon_C)}(\mathsf{A} \bowtie_{\bar{\mathsf{I}}} \mathsf{B} \bowtie_{\bar{\mathsf{I}}} \tilde{\mathsf{C}})\\
\mathsf{D}_3 &= \pi_{\check{\mathsf{I}},\mathsf{v}=f(\mathsf{A.v},\varepsilon_B,\mathsf{C.v})}(\mathsf{A} \bowtie_{\bar{\mathsf{I}}} \tilde{\mathsf{B}} \bowtie_{\bar{\mathsf{I}}} \mathsf{C})\\
\mathsf{D}_4 &= \pi_{\check{\mathsf{I}},\mathsf{v}=f(\varepsilon_A,\mathsf{B.v},\mathsf{C.v})}(\tilde{\mathsf{A}} \bowtie_{\bar{\mathsf{I}}} \mathsf{B} \bowtie_{\bar{\mathsf{I}}} \mathsf{C})\\
\mathsf{D}_5 &= \pi_{\check{\mathsf{I}},\mathsf{v}=f(\mathsf{A.v},\varepsilon_B,\varepsilon_C)}(\mathsf{A} \bowtie_{\bar{\mathsf{I}}} \tilde{\mathsf{B}} \bowtie_{\bar{\mathsf{I}}} \tilde{\mathsf{C}})\\
\mathsf{D}_6 &= \pi_{\check{\mathsf{I}},\mathsf{v}=f(\varepsilon_A,\mathsf{B.v},\varepsilon_C)}(\tilde{\mathsf{A}} \bowtie_{\bar{\mathsf{I}}} \mathsf{B} \bowtie_{\bar{\mathsf{I}}} \tilde{\mathsf{C}})\\
\mathsf{D}_7 &= \pi_{\check{\mathsf{I}},\mathsf{v}=f(\varepsilon_A,\varepsilon_B,\mathsf{C.v})}(\tilde{\mathsf{A}} \bowtie_{\bar{\mathsf{I}}} \tilde{\mathsf{B}} \bowtie_{\bar{\mathsf{I}}} \mathsf{C})\\
\mathsf{D}_8 &= \pi_{\check{\mathsf{I}},\mathsf{v}=f(\varepsilon_A,\varepsilon_B,\varepsilon_C)}(\tilde{\mathsf{A}} \bowtie_{\bar{\mathsf{I}}} \tilde{\mathsf{B}} \bowtie_{\bar{\mathsf{I}}} \tilde{\mathsf{C}})
\end{aligned}$$

(4.6) and (4.7) are represented graphically in figures 4.5a and 4.5b respectively. Disks show relations that represent the stored part of sparse arrays. Overlapping regions show relations that result from join operations.

The *Map* on two and three arrays generalises to any number m of arrays, by considering all the possible 2^m combinations of `(stored,omitted)` portions for the m arrays. Rule 4.7 describes the relational translation of a *Map* operation over m shape-aligned arrays.



(a) Two relations – (4.6). (b) Three relations – (4.7).

Figure 4.5: Graphical representation of *Map* relational translation for two and three relations respectively. Disks show relations that represent the stored part of sparse arrays. Overlapping regions represent join operations. The union of all coloured regions is the result of the *Map* operation.

Rule 4.7 (Relational mapping for *Map*).

$$\frac{\forall l \in [0;m[\colon A_l \mapsto \mathsf{A}_l \qquad B = Map(f, A_0, \ldots, A_{m-1})}{\begin{gathered} B \mapsto \mathsf{B} = \bigcup_{j=1}^{2^m} \mathsf{B}_j \\ \mathsf{B}_1 = \pi_{\bar{l}, \mathsf{v}=f(\mathsf{A}_0.\mathsf{v},\ldots,\mathsf{A}_{m-1}.\mathsf{v})}(\mathsf{A}_0 \bowtie_{\bar{l}} \cdots \bowtie_{\bar{l}} \mathsf{A}_{m-1}) \\ \mathsf{B}_2 = \pi_{\bar{l}, \mathsf{v}=f(\mathsf{A}_0.\mathsf{v},\ldots,\mathsf{A}_{m-2}.\mathsf{v},\varepsilon_{A_{m-1}})}(\mathsf{A}_0 \bowtie_{\bar{l}} \cdots \bowtie_{\bar{l}} \mathsf{A}_{m-2} \bowtie_{\bar{l}} \tilde{\mathsf{A}}_{m-1}) \\ \vdots \\ \mathsf{B}_{2^m-1} = \pi_{\bar{l}, \mathsf{v}=f(\varepsilon_{A_0},\ldots,\varepsilon_{A_{m-2}},\mathsf{A}_{m-1}.\mathsf{v})}(\tilde{\mathsf{A}}_0 \bowtie_{\bar{l}} \cdots \bowtie_{\bar{l}} \tilde{\mathsf{A}}_{m-2} \bowtie_{\bar{l}} \mathsf{A}_{m-1}) \\ \mathsf{B}_{2^m} = \pi_{\bar{l}, \mathsf{v}=f(\varepsilon_{A_0},\ldots,\varepsilon_{A_{m-1}})}(\tilde{\mathsf{A}}_0 \bowtie_{\bar{l}} \cdots \bowtie_{\bar{l}} \tilde{\mathsf{A}}_{m-1}) \end{gathered}}$$

□

This rule is valid for any combination of dense and sparse arrays. Note how the result expression simplifies to $\mathsf{B} = \mathsf{B}_1$ in case only dense arrays are involved, by applying Rule 4.2.

The exponential number of combinations generated (2^m) does not make Rule 4.7 itself impractical in terms of complexity, as there is no global solution to be found (as in the boolean satisfiability problem [Coo71]) after the simple enumeration step. Still, the complexity of the overall relational optimisation may explode with query plans involving many relations. In practice however, this it is not likely to happen, as common value for m are never larger than 3.

Aggregate

Array aggregation is supported by the operator *Aggregate*, with syntax:

$$\mathit{Aggregate}(f, j, A)$$

Assuming associative and commutative operators as defined in Table 4.4, the relational evaluation of this operator on sparse arrays can be mapped as the corresponding dense version (a standard relational aggregate), subsequently patched for all the non-stored tuples in each group. For example, if a given one-dimensional sparse array A, with $\varepsilon_A = 2$, is stored as a relation A that misses x tuples, the summation over A needs to be patched adding $2 * x$ to the result. This naturally extends to multi-dimensional arrays. Before presenting the formal mapping rule, we define binary functions f_- and f_-^+ for a number of aggregation functions:

$$\begin{aligned} f = \mathit{sum} \Rightarrow f_- &= +, \quad f_-^+ = * \\ f = \mathit{prod} \Rightarrow f_- &= *, \quad f_-^+ = \mathit{pow} \\ f = \mathit{min}/\mathit{max} \Rightarrow f_- &= \mathit{min}/\mathit{max}, \quad f_-^+ = \mathit{id}_1 \end{aligned}$$

where function id_1 always returns its first argument.

Rule 4.8 (Relational mapping for *Aggregate*).

$$\frac{A \mapsto \mathsf{A} \qquad gs = \prod_{k=0}^{j-1} \mathcal{S}_A^k \qquad B = \mathit{Aggregate}(f, j, A)}{\begin{gathered} B \mapsto \mathsf{B} = \pi_{\bar{\imath}, \mathsf{v} = f_-(G.v, f_-^+(\varepsilon A, M.v))}(\mathsf{G} \bowtie_{\bar{\imath}} \mathsf{M}) \\ \mathsf{G} = \pi_{(i_0 = i_j, \ldots, i_{n-j-1} = i_{n-1}, \mathsf{v})}\big(_{(i_j, \ldots, i_{n-1})}\mathcal{G}_{\mathsf{v} = f(\mathsf{v})} \mathsf{A}\big) \\ \mathsf{M} = \pi_{(i_0 = i_j, \ldots, i_{n-j-1} = i_{n-1}, \mathsf{v} = gs - \mathsf{v})}\big(_{(i_j, \ldots, i_{n-1})}\mathcal{G}_{\mathsf{v} = \mathsf{count}(*)} \mathsf{A}\big) \end{gathered}}$$

where gs denotes the nominal group size, and relations G, M and B compute the aggregation on stored data, the number of missing tuples per group and the final aggregation, respectively. □

ITopN

Top-N operations are not defined in standard relational algebra. However, most modern database systems implement operators to support such a functionality. SRAM assumes an extended relational algebra that includes a top-N operator τ^n, whose syntax is:

$$\tau^n[\mathit{attr1} <\uparrow \mid \downarrow> ;\mathit{attr2} <\uparrow \mid \downarrow> ; \ldots](A)$$

where n is the number of desired tuples in the result, *attr1*,*attr2*,..., are the ranking attributes, and $<\uparrow \mid \downarrow>$ denotes `<ASC|DESC>` order. If no ranking attributes are specified, a sample of up to n tuple is returned. Rule 4.9 describes the relational translation of the array operator *ITopN*(A, n,`DESC`). A similar rule exists for ascending order.

Rule 4.9 (Relational mapping for *ITopN*).

$$\frac{\begin{array}{c} A \mapsto A \\ |\mathcal{S}_A| = 1 \qquad B = \mathit{ITopN}(A, n, \mathtt{DESC}) \qquad n \leq |A| \end{array}}{\begin{array}{c} B \mapsto B = \pi_{i_0=\mathit{rowid()}, v=i_0}(\tau^n[v \downarrow](T \cup M)) \\ T = \tau^n[v \downarrow](A) \qquad M = \tau^n[\,](\tilde{A}) \end{array}}$$

First, the top n candidates are selected from stored relation A into relation T. Then, a second set of n candidates M is created for the non-stored portion of the relation, representing ε_A values. Finally, a regular top-N is performed on the union of these two set of candidates, and the index column of the result relation is projected as an array value. □

4.6 Discussion

The prototype system described in this chapter adds the missing building blocks to the layered software architecture envisioned in Chapter 3. The SRAM language allows to implement easily most of the operations of the Matrix Framework for IR. While other languages can describe sparse array computations more powerfully, like Matlab, or in a more standard data-oriented way, like SQL, SRAM is designed to bridge gracefully two domains with fundamental differences: the array domain, focused on cell positions and with a concept of order, and the relational domain, focused on unordered sets of values. A set of mapping rules is presented to provide a concrete answer to the question whether relational algebra can be used as a data-access interface to support basic linear algebra operations on sparse matrices. All elementary operations between scalars, vectors, matrices and n-dimensional arrays can be fully supported.

Iterative algorithms remain however an issue. Bounded loops over known index spaces are natively encoded in both SRAM and relational algebra as the enumeration of all possible combinations. As an example, a simple matrix multiplication requires 3 explicit loops over the index variables in the C language, while SRAM and the relational algebra encode these loops in native index-space enumeration constructs: a comprehension `[ | i,j,k ]` and a join $\bowtie$, respectively. However, the lack of a flexible looping mechanism (`while <condition> do {<body>}`) in the relational algebra, which is its fundamental limitation in expressive power, does not allow SRAM to support iterative methods based on a convergence condition, such as eigenvalue algorithms and cluster analysis. Without specific extensions of the relational algebra for looping mechanisms, the only viable approach for such class of problems is to wrap single-step operations with loop constructs of external (scripting) languages.

If everything seems under control, you're just not going fast enough.

- Mario Andretti, car racer

5 Array database optimisation

This chapter faces the problem of how to make array database queries perform efficiently. In the relational domain, query optimisation is a well-studied topic (for a survey, see [Cha98]). It is likewise known how relational query optimisation is an NP-complete problem [IK84, SM97] and how relational query optimisers are designed to aim at reasonably efficient query plans, rather than at the best possible query plan. Because array database queries translate to relational queries, they are not less hard to optimise. However, the techniques presented in this chapter aim at exploiting the peculiarities of relational queries that come from a translated array-domain. The additional knowledge of the domain in which relational technology is applied provides many handles for more focused, and hopefully more effective, query optimisation.

5.1 Array optimisation

Chapter 4 presents an algebra for array computations and shows how it can be fully mapped onto relational algebra. This suggests the possibility to ignore all optimisation issues in the array domain and shift them to the relational domain, once the translation has taken place, relying on the high availability of research results and solutions from the field of relational optimisation.

Relational query optimisation is indeed one of the benefits provided by this approach. Nevertheless, we advocate the importance of an additional optimisation phase, which operates in the array domain, for the following three reasons:

domain-contained optimisation - to keep optimisation strategies in the domain where they belong: the question should not be "is it technically feasible to move this array optimisation to the relational domain?", but rather "would it fit there?" (see Example 5.3).

representation independence - to keep array optimisations independent of relational mapping: array optimisation may map naturally to the relational domain, but only assuming a specific relational representation of arrays (see Example 5.1).

early optimisation - for efficiency: array algebra expressions are typically shorter than their relational equivalent (see Example 5.1).

5.1.1 Core algebra transformation rules

The most basic transformation rules for an array optimiser deal with the special case of constant arrays.

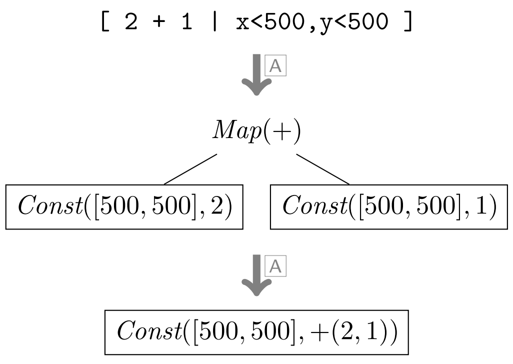
Avoid materialising constant arrays

Rule 5.1 (*Map* over *Const*). A function mapped to constant arrays is equivalent to a constant array where the function is performed just once over the constant values:

$$\frac{A = \mathit{Map}(f, \mathit{Const}(\mathcal{S}, c_0), \ldots, \mathit{Const}(\mathcal{S}, c_{m-1}))}{A \equiv \mathit{Const}(\mathcal{S}, f(c_0, \ldots, c_{m-1}))}$$

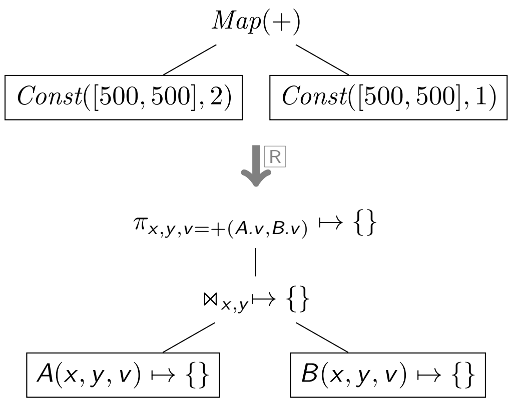
□

Example 5.1 (Map over Const).
Expression `[ 2 + 1 | x<500,y<500 ]` translates to the mapping of function '+' over two constant arrays: *Const*([500, 500], 2) and *Const*([500, 500], 1). By applying Rule 5.1, +(2, 1) is detected as a function on constant values, and used as an element value for the resulting constant array:



□

It is interesting to note that the same result would have been achieved by relying on relational optimisation only. Because each *Const* array has density $\delta = 0$, it maps to an empty relation, which makes the subsequent join and projection operators produce empty relations as well (most relational optimisers would simplify such empty relations away). This is correct, as the result of the operation is a constant array, again with density $\delta = 0$. The previous example would then be translated as follows:


$Map(+)$
$Const([500, 500], 2)$
$Const([500, 500], 1)$
R
$\pi_{x,y,v=+(A.v,B.v)} \mapsto \{\}$
$\bowtie_{x,y} \mapsto \{\}$
$A(x, y, v) \mapsto \{\}$
$B(x, y, v) \mapsto \{\}$

However, as anticipated in Section 5.1, having a native optimisation rule in the array domain has the following advantages:

- *Representation independence*: the fact that it is possible to postpone this optimisation to the relational domain obtaining the same effect relies on the relational mapping for arrays chosen in Section 4.5.1. In particular, on the the fact that sparse arrays with density $\delta = 0$ map to empty relations. On the contrary, is always possible to perform this optimisation in the array domain, where only the array properties are exploited.
- *Early optimisation*: performing the simplification at array level reduces the search space for the subsequent relational optimisation.

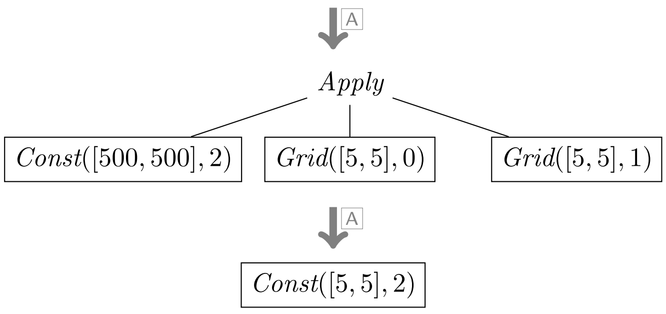
Rule 5.2 (*Apply* over *Const*). Another optimisation opportunity is the application of a constant array to a set of indices. The result is computable immediately as a constant array of the selected shape and the same constant element value:

$$\frac{A = Apply(Const(\mathcal{S}, c), I_0, \ldots, I_{k-1})}{A \equiv Const(\mathcal{S}_{I_0}, c)}$$

Recall from Section 4.4.2 that the result of an *Apply* operator has the shape of any of its index arrays, i.e. $\mathcal{S}_A = \mathcal{S}_{I_0} = \cdots = \mathcal{S}_{I_k - 1}$ □

Example 5.2 (Apply over Const).
Expression `[ 2 | x<500,y<500 ]` constructs a constant array with element value 2. Any subsequent selection on such an array is still a constant array with the same element value, which makes it possible to remove the selection and produce the correct constant array directly (e.g. `A = [ 2 | x<5,y<5 ]`):

```
A = [ 2 | x<500,y<500 ]
 [ A(x,y) | x<5,y<5 ]
```



□

The same result would be achieved by delegating this optimisation to the relational domain. However, handling the simplification in the array domain has the same advantages as in Rule 5.1: *representation independence* and *early optimisation.*

Remove identity transformations

The result of an identity transformation is by definition identical to the original. Some identity transformations are not easily identifiable. For example, using persistent arrays as index arrays for an *Apply* operator makes it impossible to predict the result, unless some additional knowledge allows for that. However, some other patterns are easily identified as identity transformations and can be simplified away.

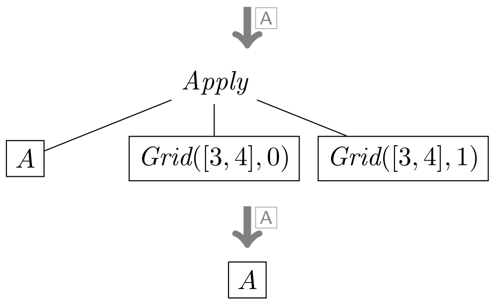
Rule 5.3 (*Apply* with identity *Grid*s). The following pattern, where index arrays are generated by the *Grid* operator, is fully predictable as an identity transformation, which can be simplified away:

$$\frac{\begin{array}{c} A' = \mathit{Apply}(A, I_0, \ldots, I_{n-1}) \\ n = |\mathcal{S}_A| \\ I_0 = \mathit{Grid}(\mathcal{S}_A, 0), \ldots, I_{n-1} = \mathit{Grid}(\mathcal{S}_A, n-1) \end{array}}{A' \equiv A}$$

□

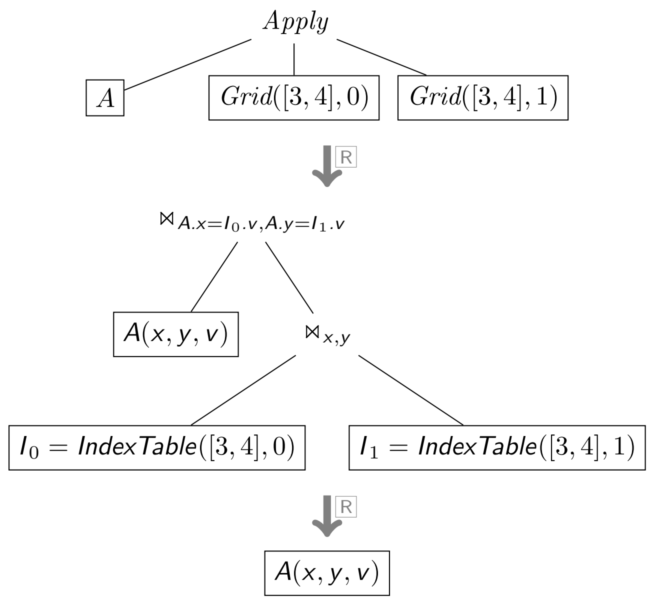
Example 5.3 (Apply with identity Grids).
Given a matrix $A[3,4]$, the following expression defines an identity transformation:

```
A = ([3,4],int) "A"
[ A(x,y) | x<3,y<4 ]
```



□

Similarly to the previous rules on constant arrays, the simplification operated by Rule 5.3 could in principle be handled in the relational domain. The expression in Example 5.3 would be translated as follows:



$Apply$
A
$Grid([3,4],0)$
$Grid([3,4],1)$
R
$\bowtie_{A.x=I_0.v, A.y=I_1.v}$
$A(x,y,v)$
$\bowtie_{x,y}$
$I_0 = IndexTable([3,4],0)$
$I_1 = IndexTable([3,4],1)$
R
$A(x,y,v)$

Postponing this rule to the relational domain would pose more fundamental issues than the previous rules do. The last step of the translation process above, that removes the join between relation A and the indices, assumes that a relational optimiser would take into account the semantics of the *IndexTable* operator, which is far from reasonable to assume. Notice that this is not in contrast with allowing the extension of a relational engine with the *IndexTable* operator. Introducing values from a different domain (the array domain in this case) into the relational domain is not a problem, as long as the original semantics of the values inserted is dropped. In other words, once they entered the relational domain, array index values become integer values, and relational optimisers should interpret them as such. In some cases it would be possible to infer properties that allow to translate the same type of reasoning from one domain into another without domain clashes. The identity transformation in the example would be straightforward to detect if matrix A were one-dimensional. In that case, the index space would result in a single dense sequence of integers. Because dense sequences of integers are understood by some relational engines (e.g. MonetDB [BK95]), these could detect a join between equal sequences as a redundant join. However, in case of multi-dimensional arrays, indices would translate to arbitrarily complex groups of sequences of integers, which are beyond the scope of any existing relational system. We can conclude that, in order to guarantee a *domain-contained optimisation* process (see Section 5.1), any optimisation that involves interpretation of array properties in the array domain, like *Grid* operators, should not be postponed to the relational domain, unless such properties can be fully translated as well.

Reduce (intermediate) result sizes

Early reduction of the result space, with the aim of reducing the data volume to be processed by each operator in the expression tree being analysed, is one of the most important optimisation principles. In the relational domain, this corresponds to pushing selective operators down the query tree, whenever possible. Similar reasoning leads to the following rule in the array domain, where an *Apply* operator can reduce the result's size, acting as a selection.

Rule 5.4 (*Apply* through *Map* (downward)). An *Apply* operator can always be pushed down through a *Map* operator, on top of all its children:

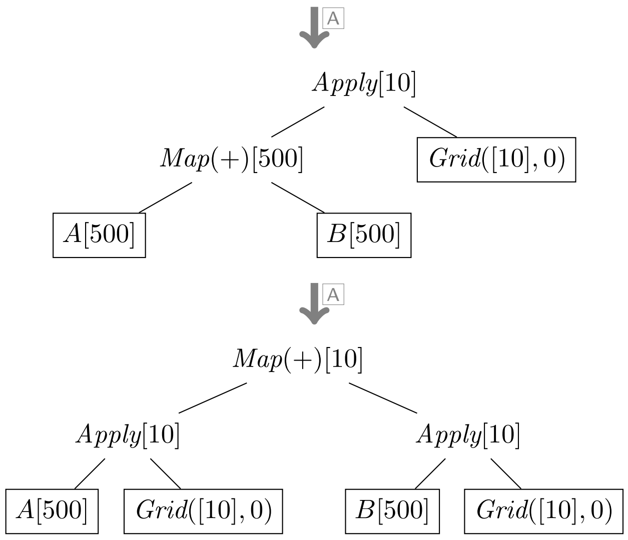
$$\frac{A' = \mathit{Apply}(\mathit{Map}(f, A_0, \ldots, A_{m-1}), I_0, \ldots, I_{n-1})}{A' \equiv \mathit{Map}(f, \mathit{Apply}(A_0, I_0, \ldots, I_{n-1}), \ldots, \mathit{Apply}(A_{m-1}, I_0, \ldots, I_{n-1}))}$$

□

Example 5.4 (Apply through Map (downward)).
This example shows how an *Apply* operator that reduces the result's shape is pushed down through a *Map* operator, which therefore needs to process smaller

arrays.

```
A = ([500], int) "A"
B = ([500], int) "B"
[ [ A(x) + B(x) | x<500 ](y) | y<10 ]
```



□

Because the *Apply* operator could however increment the result's size as well, an optimiser should consider such a transformation as bi-directional, and take into account the following rule too.

Rule 5.5 (*Apply* through *Map* (upward)). A number of *Apply* operators with identical index arrays can always be pulled up through a *Map* operator and applied only once on top of it with the same index arrays:

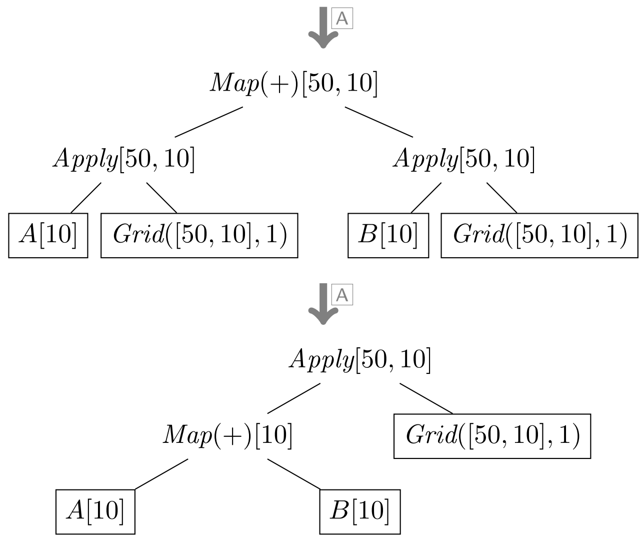
$$\frac{A' = \mathit{Map}(f, \mathit{Apply}(A_0, I_0, \ldots, I_{n-1}), \ldots, \mathit{Apply}(A_{m-1}, I_0, \ldots, I_{n-1}))}{A' \equiv \mathit{Apply}(\mathit{Map}(f, A_0, \ldots, A_{m-1}), I_0, \ldots, I_{n-1})}$$

□

Example 5.5 (Apply through Map (upward)).
This example shows how two *Apply* operators that increase the result's size are pulled up through a *Map* operator, which therefore needs to process smaller

arrays.

```
A = ([10], int) "A"
B = ([10], int) "B"
[ A(x) + B(x) | y<50,x<10 ]
```



□

Rule 5.6 (Substitution). An interesting aspect of array application is the fact that arrays are functions. For any array expression applied to indices it is possible to perform the application – i.e. to evaluate the array function – directly through substitution. In other words, it is possible to substitute an *Apply* operator that performs the extraction of elements from an array A with the array elements themselves, provided that these are deductible analytically. This is true when e.g. A is a function of *Grid* arrays, but not when A is a persistent array. Another way to describe this rule is that two consecutive definitions of the result space can be merged into one.

$$\frac{A' = \mathit{Apply}(A, I_0, \ldots, I_{n-1})}{A' \equiv \mathit{subst}(A, I_0, \ldots, I_{n-1})}$$

The auxiliary function $\mathit{subst}(A, I_0, \ldots, I_{n-1})$ replaces recursively any $\mathit{Grid}(\mathcal{S}_A, i)$ in A by I_i. In order to substitute only the latest (thus up in the expression tree) shape definition for A, the recursion of function *subst* is limited as follows. At each node in A's subtree, it only propagates to the children that may be directly responsible for defining the shape of A by means of *Grid* operators, or to those children that do not alter the shape. For example, the recursion would be propagated to the index arrays of an *Apply* operator in A's subtree (not to its first child though, which has no role in the result's shape definition), or to the children of a *Map* operator (that cannot modify shapes). However, it would not be propagated to the children of operators like *Aggregate*, that do alter the shape of their children by other means than *Grid* operators (thus, not in a predictable way). □

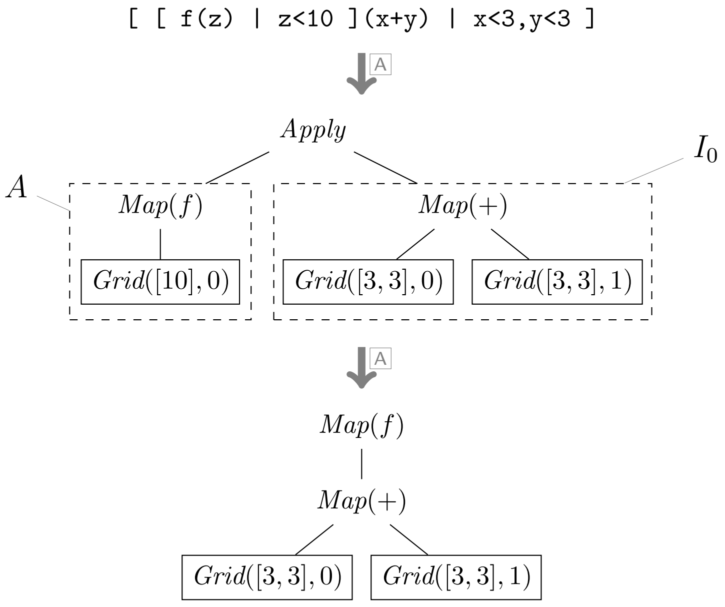
Example 5.6 (Substitution).
In the following expression,

```
[ [ f(z) | z<10 ](x+y) | x<3,y<3 ]
```

the index `z` can be substituted and the result obtained directly:

```
[ f(x+y) | x<3,y<3 ]
```

Such a simplification is detected and performed by an array algebra optimiser applying Rule 5.6 (in dashed boxes, the patterns that are recognised as A and I_0 in the rule):



□

Rule 5.7 (Independent axes). This rule can be applied to an array whose element values are independent of some of its axes. The element value can be evaluated over only the axes it depends on and extended afterwards to the desired shape.

$$\frac{\begin{array}{c} A \\ U = \mathit{set2array}(\{j \mid \mathit{Grid}(\mathcal{S}, j) \text{ is in } A\text{'s tree}\}) \\ l = |U| \qquad l < |\mathcal{S}_A| \end{array}}{\begin{array}{c} A \equiv \mathit{Apply}(A', \mathit{Grid}(\mathcal{S}_A, U(0)), \ldots, \mathit{Grid}(\mathcal{S}_A, U(l-1))) \\ A' = \mathit{subst}(A, \mathcal{S}', U) \\ \mathcal{S}' = [\mathcal{S}_A(U(0)), \ldots, \mathcal{S}_A(U(l-1))] \end{array}}$$

where:

$\mathit{set2array}()$ sorts the elements of a set in ascending order, producing an array

$\mathit{subst}(A, \mathcal{S}', U)$ replaces recursively any $\mathit{Grid}(\mathcal{S}, i)$ in A by $\mathit{Grid}(\mathcal{S}', \mathit{pos}(i, U))$

$\mathit{pos}(i, U)$ returns the position of element i in vector U

This transformation rule can be described more informally as follows. Given an array A, if the axes used for its definition (U) are fewer than its valence ($|\mathcal{S}_A|$), then the definition of A can be decomposed in two steps:

1. *subst*() restricts the shape of A to $\mathcal{S}'$, that is identified by U. This step evaluates only the elements of A that depend on U.

2. *Apply*() replicates the result of the previous step for each value of the axes that are not in U, yielding the original shape $\mathcal{S}_A$. Recall that when an axis is not used in the definition of an array element, this element is the same for each value of that axis. □

Example 5.7 (Independent axes).
The expression

```
[ f(x) | x<3,y<4 ]
```

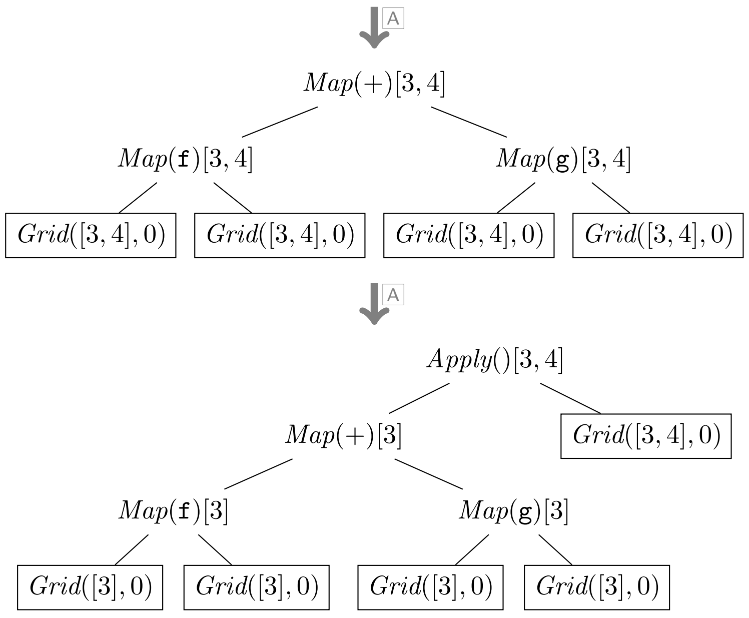
defines a bi-dimensional array of shape $[3, 4]$ with axes `x` and `y`. However its element values depend only on the value of the `x` axis. If `f` is an expensive function, it may be cheaper to re-formulate the expression above as follows:

```
[ [ f(x) | x<3 ](x) |x<3,y<4 ],
```

where `f` is first evaluated for all values of `x` and the result subsequently replicated for all values of `y`.

Applying Rule 5.7, the same rewriting can be performed on the equivalent algebra expression:

```
[ f(x) + g(x) | x<3,y<4 ]
```



□

5.1.2 Reusing indices: the input space approach

The *shape alignment* normalisation step for SRAM syntax, described in Section 4.4.1, emphasises the definition of arrays as functions of their own result index domain (see Definition 4.5). This normalisation step has the effect of reshaping every element in an SRAM expression to the index domain of the expression itself. Such induced reshaping operations, together with those explicitly entered by the user, translate to array algebra expressions that involve the combination of *Apply* and *Grid* operators. The advantage of such a combination is that it captures a wide spectrum of array manipulations (e.g. reshaping, replication, index-based selection) in an unified manner. However, one important drawback of this approach can be a loss of explicit information on the original array expression, which may in turn result in translation patterns that are too generic to be efficient. For instance, the translation

```
 A = ([300,400],dbl) "A"
A' = [ A(y,x) | x<400,y<300 ]
```

$$\Big\downarrow\boxed{\text{A}}$$

$$A' = \mathit{Apply}(A, \mathit{Grid}([400, 300], 1), \mathit{Grid}([400, 300], 0))$$

makes the original matrix transposition pattern be blotted out by a generic *Apply* + *Grid* combination.

In practice, the "*Apply* + *Grid* approach" suffers from two problems:

1. the *Apply* operator, in combination with *Grid*, covers too many functionalities, such as shuffling array contents, zooming in on a small part of an array, or replicating values for shape alignment with a different array. This complicates the mapping onto highly optimised expressions in the target language;

2. the operator *Grid* produces by definition only dense index arrays. Unfortunately, the generic processing style through *Apply* and *Grid* operators is not well-suited for the efficient handling of sparse arrays, although still correct. In general, the indices that address only the stored part (non-ε values) of a sparse array do not follow a regular pattern, and therefore could not be generated by a *Grid*-like operator. When used in combination with *Apply* to index a sparse array, the generation of indices through *Grid* operators may nullify the benefits of the compressed storage scheme for sparse arrays, and result in a considerable waste of system resources, often not acceptable in practice.

The challenge in the efficient evaluation of expressions involving sparse arrays is to use a compact representation of data during all the intermediate computation steps. The *input space approach* described in this section aims to achieve this goal by directly manipulating existing indices whenever possible, thus only using the stored portion of sparse arrays and minimising the need for potentially very expensive *Grid* operations.

This is possible through the addition of new array operators to the core set presented in Section 4.4, that are equivalent to specific *Apply* + *Grid* patterns:

Pivot$(A, \mathcal{P})$ permutes the dimensions of A following the axis order permutation specified in $\mathcal{P}$ (e.g. $\mathcal{P} = [1, 0]$ for matrix transposition).

RangeSel$(A, \mathcal{O}, \mathcal{S})$ selects from A the sub-array identified by offset $\mathcal{O}$ from the origin and shape $\mathcal{S}$.

Replicate(A, k) increases the number of dimensions of A by 1, by replicating it k times.

JoinMap$(f, \mathcal{J}, A_0, \ldots, A_{m-1})$ maps the function f to corresponding elements of arrays $A_0, \ldots, A_{m-1}$. $\mathcal{J}$ indicates the list of axes that $A_0, \ldots, A_{m-1}$ have in common.

This list may of course be further expanded with additional specialised operators that are expected to yield efficient relational query plans for specific patterns. Interesting examples include different selection patterns, such as for selecting diagonals, and complex operators that would e.g. support MapReduce algorithms [DG04, LD10], facilitating the efficient execution of split, map, join and aggregate operations there involved.

Such operators are introduced by the optimiser by rewriting expressions that use core array operators (see Table 4.2) into more specialised operators, and never directly derived from the SRAM syntax. In other words, the input space approach forms a second optimisation phase, after the application of the transformation rules presented in Section 5.1.1, that is specialised for handling the sparse arrays compressed representation. This has the following advantages:

- dense and sparse arrays share a common first optimisation phase, that is based on transformation rules defined over a compact set of operators;
- the clean direct relationship among the functional definition of arrays, the SRAM syntax and the core array algebra is preserved. This provides a more theoretically solid ground for the definition and the application of the transformation rules presented in Section 5.1.1.

Minimising the usage of *Grid* operators is especially beneficial for processing sparse arrays. The underlying assumption is that the physical representation of a sparse array may be several orders of magnitude smaller than its nominal size. Given a sparse array that fits well in the physical resources available, its dense representation, and *Grid* operators in the same nominal size, may cause a severe waste of the same physical resources or even exceed them. *Grid* operators are less critical in the evaluation of dense arrays, as they use physical resources in the same order of magnitude of the arrays being evaluated, which are supposed to be reasonably sized. However, the input space oriented approach may be beneficial for evaluating dense arrays too, even though to a possibly smaller extent. The potential benefit depends on the problem at hand and the choice to introduce these operators is determined by the optimiser.

Splitting *Apply* operations

One of the advantages of the "*Apply* + *Grid*" combination is that consecutive occurrences of such a pattern can be merged into one single operation, thanks to the functional (thus predictable analytically) nature of the *Grid* operator (we focus here on the *Grid* operator, but for the same reason, patterns including the *Const* operator qualify as well for such a merging). For example, given array $A[3]$, the two reshaping operations

$$\begin{aligned} A' &= \mathit{Apply}(A, \mathit{Grid}([4,3],1)) && \text{(replication)} \\ B &= \mathit{Apply}(A', \mathit{Grid}([3,4],1), \mathit{Grid}([3,4],0)) && \text{(pivoting)} \end{aligned}$$

can be merged into a single one:

$$B = \mathit{Apply}(A, \mathit{Grid}([3,4],0))$$

The inverse transformation, that is to split *Apply* operations into a series of more elementary ones, is also possible. Although it does not provide any direct benefit in terms of optimisation, this second transformation may simplify the detection of elementary reshaping operations like the ones described by *Pivot*, *RangeSel* and *Replicate* operators.

Rule 5.8 (Splitting *Apply* operations). This rule states that it is always possible to split *Apply* operations in two steps (when no simpler transformations are deductible analytically, these two steps are the original transformation and an identity transformation):

$$\frac{B = \mathit{Apply}(A, I_0, \ldots, I_{n-1})}{\begin{array}{c} \exists\; I'_0, \ldots, I'_{n-1}, \quad \exists\; I''_0, \ldots, I''_{k-1} : B' \equiv B \\ B' = \mathit{Apply}(\mathit{Apply}(A, I'_0, \ldots, I'_{n-1}), I''_0, \ldots, I''_{k-1}) \end{array}}$$

Note that the split transformation may be not unique: several combinations of $I'_0, \ldots, I'_{n-1}$ and $I''_0, \ldots, I''_{k-1}$ may exist that make the equivalence true (while the inverse transformation is unique). For this reason, it is not possible to provide a set of rules that cover all patterns. An array optimiser can implement the detection of a number of patterns, triggered by heuristic strategies aimed at achieving specific goals (e.g. translating *Apply* into combinations of *Pivot*, *RangeSel* and *Replicate*). □

In the following, the operators *Pivot*, *RangeSel*, *Replicate* and *JoinMap* are described, including the rewriting rules to generate them from core algebra and the rules to map them onto relational algebra.

Pivot

The *Pivot* operator captures *Apply*+ *Grid* combinations that perform axis transposition operations.

Rule 5.9 (*Apply* to *Pivot*).

$$\frac{\begin{array}{c} B = \mathit{Apply}(A, \mathit{Grid}(\mathcal{S}, j_0), \ldots, \mathit{Grid}(\mathcal{S}, j_{n-1})) \\ [\mathcal{S}(j_0), \ldots, \mathcal{S}(j_{n-1})] = \mathcal{S}_A \end{array}}{B \equiv \mathit{Pivot}(A, [j_0, \ldots, j_{n-1}])}$$

The premise $[\mathcal{S}(j_0), \ldots, \mathcal{S}(j_{n-1})] = \mathcal{S}_A$ makes the pattern qualify for the transformation rule when the shape $\mathcal{S}$ is a permutation of the original shape $\mathcal{S}_A$, and that the permutation order is identified by integers $[j_0, \ldots, j_{n-1}]$ $\square$

The relational mapping for *Pivot* uses a projection operation to exchange the order of index columns in the original relation.

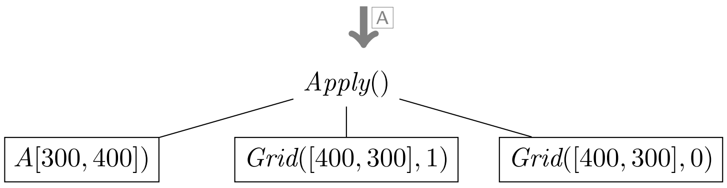
Rule 5.10 (Relational mapping for *Pivot*).

$$\frac{B = \mathit{Pivot}(A, [j_0, \ldots, j_{n-1}]) \qquad A \mapsto \mathsf{A}}{B \mapsto \mathsf{B} = \pi_{\mathsf{i}_{j_0}, \ldots, \mathsf{i}_{j_{n-1}}, \mathsf{v}}(\mathsf{A})}$$

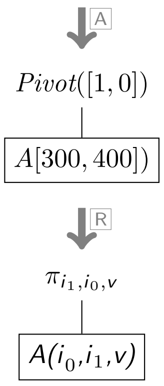
$\square$

Example 5.8 (Sparse matrix transposition).
Consider the following sparse matrix transposition

```
A = ([300,400],dbl) sparse(0,0.1) "A"
     A’ = [ A(y,x) | x<400,y<300 ]
```



This is first translated into a *Pivot* operation, and then mapped onto relational algebra with a project operation:



□

RangeSel

The *RangeSel* operator performs range selections over arrays. Given an array A, it zooms in on a sub-array identified by an offset $\mathcal{O}$ from the origin and a shape $\mathcal{S}$, that together define a multi-dimensional index range. This operator is equivalent to a pattern that uses an *Apply* operator together with optional *Const* operators to define the offset $\mathcal{O}$, *Grid* operators to define the index domain identified by the new shape $\mathcal{S}$ and *Map* operators to sum the previous two.
Rule 5.11 (*Apply* to *RangeSel*).

$$\frac{\begin{array}{c} B = \mathit{Apply}(A, I_0, \ldots, I_{n-1}) \\ |B| < |A| \qquad |\mathcal{S}_B| = |\mathcal{S}_A| \\ \forall k \in [0, n[\quad : \quad \bigvee \begin{cases} I_k = \mathit{Grid}(\mathcal{S}, k) \\ I_k = \mathit{Map}(+, \mathit{Const}(\mathcal{S}, c_k), \mathit{Grid}(\mathcal{S}, k)) \end{cases} \qquad |\mathcal{S}| = |\mathcal{S}_A| \end{array}}{\begin{array}{c} B \equiv \mathit{RangeSel}(A, \mathcal{O}, \mathcal{S}) \\ \forall k \in [0, n[\quad : \quad \mathcal{O}(k) = \begin{cases} 0 & \text{if } I_k = \mathit{Grid}(\mathcal{S}, k) \\ c_k & \text{if } I_k = \mathit{Map}(+, \mathit{Const}(\mathcal{S}, c_k), \mathit{Grid}(\mathcal{S}, k)) \end{cases} \end{array}}$$

□

The relational mapping for *RangeSel* involves two steps:

1. (σ) perform the selection of tuples whose index attributes satisfy the range condition imposed by $\mathcal{O}$ and $\mathcal{S}$;
2. (π) shift the selected sub-array to the origin (in accordance with Definition 4.5).

Rule 5.12 (Relational mapping for *RangeSel*).

$$\frac{B = \mathit{RangeSel}(A, \mathcal{O}, \mathcal{S}) \qquad A \mapsto \mathsf{A}}{\begin{array}{c} B \mapsto \mathsf{B} \\ \mathsf{B} = \pi_{\mathsf{i}_0 = \mathsf{i}_0 - \mathcal{O}(0), \ldots, \mathsf{i}_{n-1} = \mathsf{i}_{n-1} - \mathcal{O}(n-1), \mathsf{v}}(\mathsf{X}) \\ \mathsf{X} = \sigma_{\mathsf{i}_0 \geq \mathcal{O}(0) \wedge \cdots \wedge \mathsf{i}_{n-1} \geq \mathcal{O}(n-1) \wedge \mathsf{i}_0 < \mathcal{S}(0) + \mathcal{O}(0) \wedge \cdots \wedge \mathsf{i}_{n-1} < \mathcal{S}(n-1) + \mathcal{O}(n-1)}(\mathsf{A}) \end{array}} \qquad \square$$

Replicate

The operator *Replicate* increases by one the number of dimensions (valence) of an array, by replicating it k times. The new dimension, that becomes the first one in the new array, is long k.

Rule 5.13 (*Apply* to *Replicate*).

$$\frac{\begin{array}{c} B = \mathit{Apply}(A, \mathit{Grid}(\mathcal{S}, 0), \ldots, \mathit{Grid}(\mathcal{S}, n-1)) \\ |\mathcal{S}| = |\mathcal{S}_A| + 1 \\ [\mathcal{S}(1), \ldots, \mathcal{S}(n)] = \mathcal{S}_A \qquad k = \mathcal{S}(0) \end{array}}{B \equiv \mathit{Replicate}(A, k)} \qquad \square$$

The relational mapping for *Replicate* performs a cartesian product between the enumeration of the new axis values and the original array.

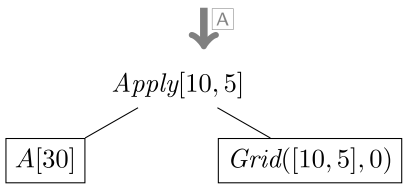
Rule 5.14 (Relational mapping for *Replicate*).

$$\frac{B = \mathit{Replicate}(A, k) \qquad A \mapsto \mathsf{A}}{B \mapsto \mathsf{B} = \mathsf{IndexTable}([k]) \times \mathsf{A}} \qquad \square$$

Although a potentially large $\mathsf{IndexTable}()$ operator is used in this mapping, the equivalent based on the *Apply*+ *Grid* combination materialises a *Grid* in the size of result array B and then produces the result, which needs nearly twice as much system resources.

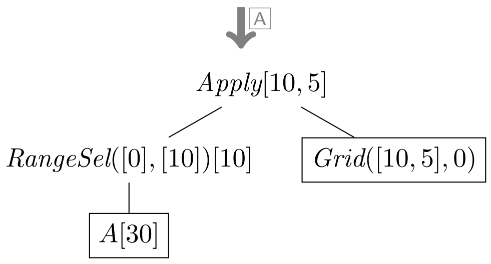
Example 5.9 (Splitting a complex Apply + Grid combination).
Consider the following array transformation:

```
A = ([30],int) sparse(0,0.1) "A"
    A' = [ A(x) | x<10,y<5 ]
```

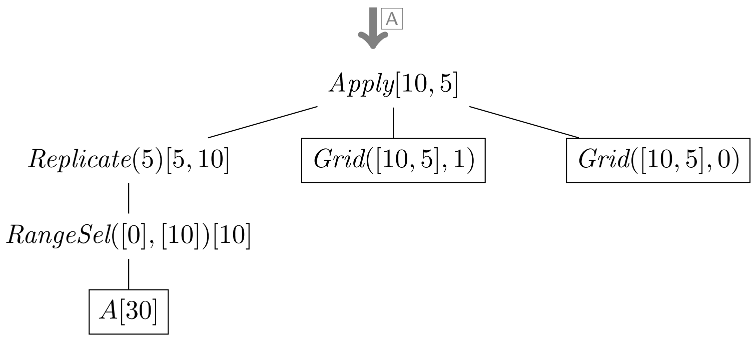


It performs a range selection on axis x and replicates the result 5 times along a new dimension y. This example shows a possible transformation of this *Apply* + *Grid* combination into a sequence of more specialised operators.

First, a *RangeSel* operator is extracted out of the original *Apply*, which yields an array of shape [10]. The residual *Apply* still needs the same *Grid* as before: *Grid*([10, 5], 0). However, no more selection is needed on the new input.



The next operator extracted from the residual *Apply* is *Replicate*:



The *Replicate* operator creates a new major axis of length 5, which yields a shape $[5, 10]$. The residual *Apply* must then swap the two axes, in order to produce the desired shape $[10, 5]$ (note that because the new intermediate result is bi-dimensional, two *Grid* operators are now needed). The last *Apply* matches indeed the pattern required for a *Pivot* transformation:

$$\begin{tikzcd}
{} \arrow[d, "\boxed{\text{A}}"] \\
\textit{Pivot}([1,0])[10,5] \arrow[d, dash] \\
\textit{Replicate}(5)[5,10] \arrow[d, dash] \\
\textit{RangeSel}([0],[10])[10] \arrow[d, dash] \\
\boxed{A[30])}
\end{tikzcd}$$

□

JoinMap

Operators like *Pivot*, *RangeSel* and *Replicate* may handle their input array more efficiently than their *Apply*-based equivalent. Further improvements are possible, however, when more than one input array are to be processed by the same operator. Such an opportunity arises with the *Map* operator.

Recall that, because of the *shape-alignment* normalisation step described in Section 4.4.1, every element of an SRAM expression is reshaped as to match the shape of the result array. In a $Map(f, A_0, \ldots, A_{m-1})$ expression, all the $A_0, \ldots, A_{m-1}$ arrays are expected to match the *Map*'s result shape after reshaping. Combinations of *Pivot*, *RangeSel* and *Replicate* operators may be estimated to perform such reshaping operations more efficiently than their equivalent expressions based on the *Apply* operator. However, the resulting *Map* expression may still translate into a sub-optimal relational plan when considered as a whole (more details follow):

lazy shape alignment - reshaping the $A_k, k \in [0, m[$ arrays may create large intermediate results before the actual processing of the *Map* operator. These could be avoided: the correct result can be produced starting from the stored index values of all the A_k arrays, also when these are not shape-aligned. This "lazy" approach to shape alignment aims at minimising expensive intermediate result materialisations by operating directly on non shape-aligned arrays;

early filtering of non-qualifying indices - the performance loss caused by a "short-sighted" reshaping of the A_k arrays (i.e. unaware of the *Map* expression as a whole) has a dramatically larger impact when sparse arrays are to be processed: the lower the input arrays' density, the the lower the chances of materialised indices in the result array.

The examples shown hereafter consider *Map* operations between two arrays A and B, and their possible rewriting into newly introduced *JoinMap* operations. The mappings onto relational plans refer to dense arrays or to sparse arrays with $\varepsilon_A = \varepsilon_B = 0$. In both this cases we are allowed to reduce the result of 4.6 down to expression $C_1 = \pi_{\bar{I},v=f(A.v,B.v)}(A \bowtie_{\bar{I}} B)$, which simplifies the presentation of the concepts in this section (that are however valid for any expression).

Consider the following three expressions:

arrays A and B are both aligned to the result array already:

```
[ A(x,y) * B(x,y) | x,y ]
```
(5.1)

arrays A and B share one of their axes (y):

```
[ A(x,y) * B(y,z) | x,y,z ]
```
(5.2)

arrays A and B share no axis:

```
[ A(x,y) * B(w,z) | x,y,w,z ]
```
(5.3)

One way to explain what these three expressions have in common is that the elements of A and B that are to be multiplied are functions of the result array index domain. Such index domains are identified by all the combinations of values for the axes in the comprehension expression: `x,y` for (5.1), `x,y,z` for (5.2) and `x,y,w,z` for (5.3). In array algebra, the enumeration of such combinations is obtained "per-array", that is reshaping each array as mentioned above, due to the shape-alignment normalisation step. We can see such an approach depicted in Figure 5.1 for the three expressions. Notice that the array algebra translation shown there is already partially optimised with *Pivot* and *Replicate* operators in place of *Apply* and *Grid* operators. Nevertheless, such an optimisation does not remove the main disadvantage of this approach: each array is replicated up to the result's index domain.

However, in the three expressions above, all the axis combinations can be

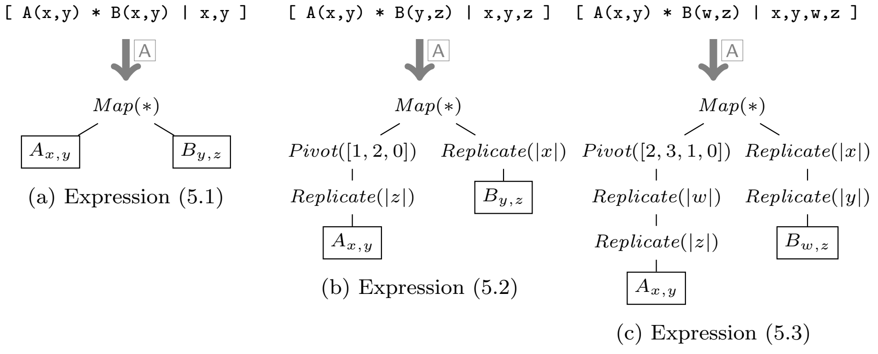


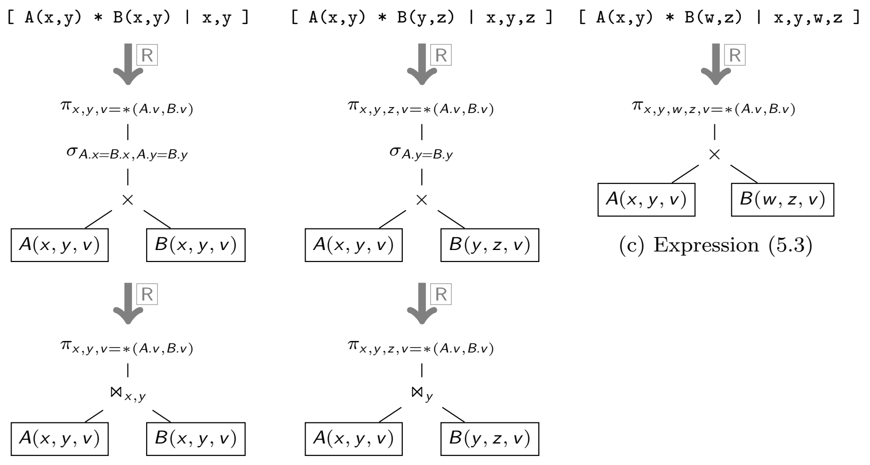
Figure 5.1: The standard array algebra translation and optimisation for three different *Map* operations

obtained "collaboratively", each array contributing the axis values that it contains. In relational algebra, obtaining all the combinations from values stored in different tables translates to using the cartesian product operator. On top of those combinations, the three expressions differ as follows: in (5.1), axes `x` and `y` have the same value for each combination in both arrays (only the combinations that satisfy this condition should be retained); in (5.2), only axis `y` is shared between the two arrays (only the combinations that satisfy this condition should be retained); in (5.3), arrays A and B share no axis (all the combinations should be considered). The restrictions to the enumeration of axis combinations of (5.1) and (5.2) remove duplicates that would be created due to the existence of the same axis values in more than one array. In relational terms, such restrictions translate into joins: selections on top of the cartesian products. Figure 5.2 shows this approach on the three expressions discussed above.

Such relational translations, compared to the ones derivable from array expressions as in Figure 5.1, allow for the achievement of the two advantages anticipated at the beginning of this section:

lazy shape alignment - the result index domain is created only once, using the index domains of arrays A and B, rather than three times (one for array A, one for array B and one for the result array). This is possible thanks to the relational cartesian product operator. The impact of this optimisation is equal for both dense and sparse array processing.

early filtering of non-qualifying indices - A selection on top of the


[A(x,y) * B(x,y) | x,y]
[A(x,y) * B(y,z) | x,y,z]
[A(x,y) * B(w,z) | x,y,w,z]
R
$\pi_{x,y,v=*(A.v,B.v)}$
$\sigma_{A.x=B.x,A.y=B.y}$
$\times$
$A(x,y,v)$
$B(x,y,v)$
$\bowtie_{x,y}$
$\pi_{x,y,z,v=*(A.v,B.v)}$
$\sigma_{A.y=B.y}$
$B(y,z,v)$
$\bowtie_{y}$
$\pi_{x,y,w,z,v=*(A.v,B.v)}$
$B(w,z,v)$

(a) Expression (5.1) (b) Expression (5.2) (c) Expression (5.3)

Figure 5.2: The desired relational translation for three different *Map* operations

cartesian product operator filters out the index combinations that do not qualify as belonging to the result index domain. However, these get materialised before such a selection. Thanks to the relational join operator, the selection is embedded in the cartesian product, so that combinations that do not satisfy the selection criteria do not get materialised. The impact of this optimisation step can be extremely high, especially when processing sparse arrays: the lower the density of arrays A and B, the lower the chance to find tuples from the resulting tables that match the join conditions (that become therefore more selective).

Summarising the three cases above, a new *JoinMap* array operator can be introduced as a generalisation of the *Map* operator, to allow direct handling of non shape-aligned arrays. This new operator captures patterns that can be easily translated into efficient relational expression that use the join operator and no *Apply* + *Grid* combinations.

Rule 5.15 (*Map* to *JoinMap*). The pattern captured by this rule is a *Map* on top of arrays that are to be reshaped to the result's shape, by means of *Apply* + *Grid* combinations. Arrays that need no reshaping, because they are shape-aligned to the result array already, can always be interpreted as *Apply* + *Grid* combinations that result in identity transformations. For this reason, the rule can assume all the $A_0, \ldots, A_{m-1}$ arrays are *Grid* applications.

$$\frac{\begin{array}{c} B = \mathit{Map}(f, A_0, \ldots, A_{m-1}) \\ \forall k \in [0, m[: A_k = \mathit{Apply}(A'_k, I_{k0}, \ldots, I_{k|\mathcal{S}_{A'_k}|-1}) \\ \forall l \in [0, |\mathcal{S}_{A'_k}|[: I_{kl} = \mathit{Grid}(\mathcal{S}_{A_k}, j_{kl}) \end{array}}{\begin{array}{c} B \equiv \mathit{JoinMap}(f, \mathcal{J}, A'_0, \ldots, A'_{m-1}) \\ \mathcal{J} = [\mathcal{J}_0, \ldots, \mathcal{J}_{|L|-1}] \\ \forall l \in [0, |L|[: \mathcal{J}_l = [\mathit{pos}(L(l), L_0), \ldots, \mathit{pos}(L(l), L_{m-1})] \\ L = L_0 \cap \cdots \cap L_{m-1} \\ \forall k \in [0, m[: L_k = [j_{k0}, \ldots, j_{k|\mathcal{S}_{A'_k}|-1}] \end{array}}$$

where $\mathit{pos}(x, X)$ returns the 0-based position of element x in vector X.
The result is a *JoinMap* expression that operates directly on the $A'_0, \ldots, A'_{m-1}$ arrays (thus skipping all the *Grid* applications). The additional knowledge that this operator needs to carry in order to be processed in relational algebra is a list of the axes that are shared among the $A'_0, \ldots, A'_{m-1}$ arrays and the result array. This knowledge is encoded as vector of vectors, $\mathcal{J}$. For each shared axis l, $\mathcal{J}(l) = \mathcal{J}_l$ lists the positions of that axis in each of the $A'_0, \ldots, A'_{m-1}$ arrays. This determines which table attributes will form the join conditions in the relational translation. □

Rule 5.16 (Relational mapping for *JoinMap*). The relational mapping for *JoinMap* is similar to the one for *Map* (see Rule 4.7), except that the join conditions do not entail all the index attributes, but only those listed by $\mathcal{J}$.

$$\frac{\begin{array}{c} \forall l \in [0; m[: A_l \mapsto \mathsf{A}_l \\ B = \mathit{JoinMap}(f, \mathcal{J}, A_0, \ldots, A_{m-1}) \end{array}}{\begin{array}{c} B \mapsto \mathsf{B} = \bigcup_{j=1}^{2^m} \mathsf{B}_j \\ \mathsf{B}_1 = \pi_{\mathcal{J}, \mathsf{v}=f(\mathsf{A}_0.\mathsf{v}, \ldots, \mathsf{A}_{m-1}.\mathsf{v})}(\mathsf{A}_0 \bowtie_{\mathcal{J}} \cdots \bowtie_{\mathcal{J}} \mathsf{A}_{m-1}) \\ \mathsf{B}_2 = \pi_{\mathcal{J}, \mathsf{v}=f(\mathsf{A}_0.\mathsf{v}, \ldots, \mathsf{A}_{m-2}.\mathsf{v}, \varepsilon_{A_{m-1}})}(\mathsf{A}_0 \bowtie_{\mathcal{J}} \cdots \bowtie_{\mathcal{J}} \mathsf{A}_{m-2} \bowtie_{\mathcal{J}} \tilde{\mathsf{A}}_{m-1}) \\ \vdots \\ \mathsf{B}_{2^m-1} = \pi_{\mathcal{J}, \mathsf{v}=f(\varepsilon_{A_0}, \ldots, \varepsilon_{A_{m-2}}, \mathsf{A}_{m-1}.\mathsf{v})}(\tilde{\mathsf{A}}_0 \bowtie_{\mathcal{J}} \cdots \bowtie_{\mathcal{J}} \tilde{\mathsf{A}}_{m-2} \bowtie_{\mathcal{J}} \mathsf{A}_{m-1}) \\ \mathsf{B}_{2^m} = \pi_{\mathcal{J}, \mathsf{v}=f(\varepsilon_{A_0}, \ldots, \varepsilon_{A_{m-1}})}(\tilde{\mathsf{A}}_0 \bowtie_{\mathcal{J}} \cdots \bowtie_{\mathcal{J}} \tilde{\mathsf{A}}_{m-1}) \end{array}}$$

where

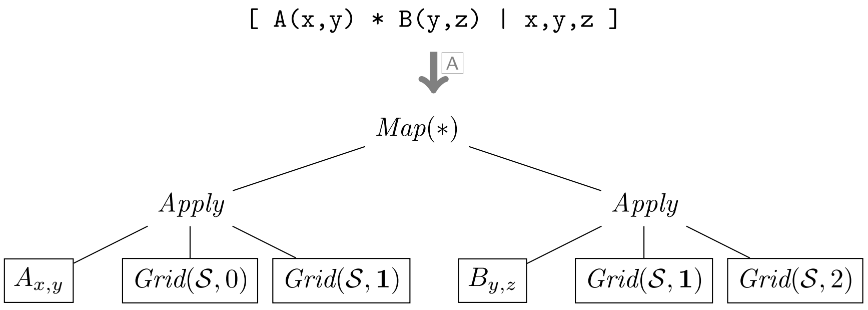
$$\bowtie_{\mathcal{J}} = \bowtie_p,$$
$$p = \bigwedge_{l=0}^{|J|-1} \left(\bigwedge_{k=0}^{m-2} \left(i_{\mathcal{J}(l)(k)} = i_{\mathcal{J}(l)(k+1)} \right) \right)$$

and

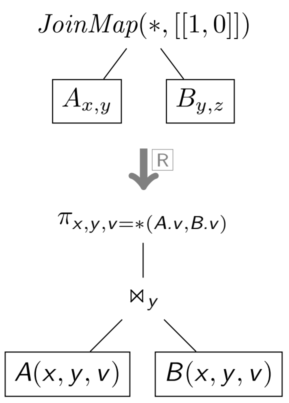
$$\pi_{\mathcal{J}} = \pi_q,$$
$$q = i_{\mathcal{J}(0)(0)}, \ldots, i_{\mathcal{J}(|J|-1)(0)}$$ □

Now we can reformulate in terms of *JoinMap* the three expressions previously seen, obtaining the transformations shown in Figure 5.3. The second transformation is shown hereafter in detail, to illustrate Rule 5.16.

Example 5.10 (Map to JoinMap).
This example shows the complete transformation path of the expression in (5.2). The standard array algebra expression contains a *Map* on top of two arrays A and B, reshaped through *Grid* applications. Rule 5.16 transform such a pattern into a *JoinMap* operator. In the following, only the symbol $\mathcal{S}$ is shown in place of the result's shape $\mathcal{S} = [|x,|,|y|,|w|,|z|]$, for presentation purposes.



Application of Rules 5.15 and 5.16 yields an optimal relational plan for this array pattern:



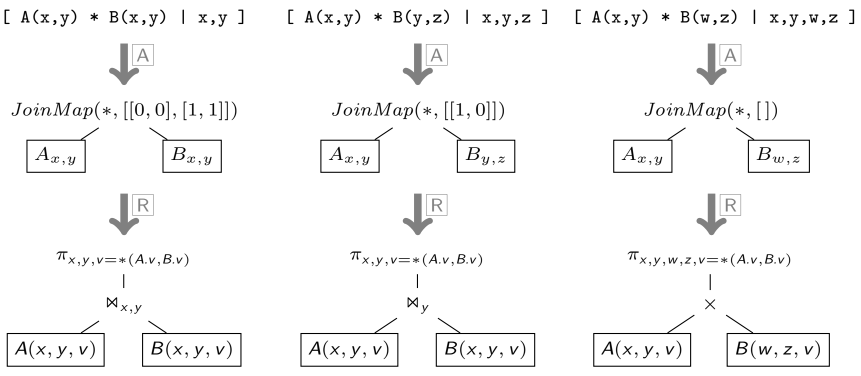
The only axis that is shared between A's and B's reshaping applications is the one with order number 1 in the result's shape, thus axis $\mathtt{y}$. Vector $\mathcal{J}$ contains only one vector with the position of axis $\mathtt{y}$ in arrays A and B. Formally, the rule is applied to this example as follows:

$$\frac{\begin{array}{c} C = Map(*, C_0, C_1) \\ C_0 = Apply(A, Grid(\mathcal{S}, 0), Grid(\mathcal{S}, 1)) \\ C_1 = Apply(B, Grid(\mathcal{S}, 1), Grid(\mathcal{S}, 2)) \end{array}}{\begin{array}{c} C \equiv JoinMap(*, \mathcal{J}, A, B) \\ \mathcal{J} = [\mathcal{J}_0] \\ \mathcal{J}_0 = [pos(1, L_0), pos(1, L_1)] = [1, 0] \\ L = L_0 \cap L_1 = [1] \\ L_0 = [0, 1] \qquad L_1 = [1, 2] \end{array}} \qquad \square$$

5.1.3 Array fragmentation

Identification of suited fragmentation patterns for given data and query expressions is a prerequisite for a number of advanced execution strategies. For example, fragmentation can be employed to limit the consumption of system resources, by dealing with only one sub-problem at the time. When multiple processing units are available, several fragments can be evaluated in parallel, provided that they do not have inter-dependencies. The same considerations can be extended to a distributed environment, with processing units connected by a network.

Data fragmentation strategies can be divided into two main classes: static



(a) Expression (5.1) (b) Expression (5.2) (c) Expression (5.3)

Figure 5.3: The desired relational translation for three different *Map* operations, using the *JoinMap* algebra operator

data fragmentation and query-driven fragmentation. The former tries to derive fragmentation patterns that can accommodate well the query patterns expected in the context of the application accessing such data, and makes those fragments persistent on disk. The main cost factors involved in such a scenario have been well studied in the context of distributed relational databases, e.g. see [CP85]. The latter tries to adjust the fragmentation strategy to each query at hand. Each derived fragment can be made persistent on disk, which realises an incremental fragmentation when more similar queries are issued on the same data (see e.g. [IKM09]), or forgotten at each query, when the queries to be issued are not expected to benefit from their mutual induced data fragmentation.

Arrays are in principle well suited for the identification of fragmentation patterns. In particular, fully predictable fragmentation patterns in terms of interdependencies, costs and evenly distributed workloads can be identified for dense arrays [vBCdVK05]. This is not the case for sparse arrays, especially when data are distributed non-uniformly, which makes the impact of fragmentation less predictable.

We consider non-incremental query-driven data fragmentation only, achieved by fragmenting the result space of complex array algebra expressions.

5.1.4 Result space fragmentation

Disjunct algebra sub-expressions are independent from each other: in the expression $f(E_A, E_B)$, sub-expressions E_A and E_B have no side effects and can potentially be evaluated in parallel. This makes each argument of any algebra operator a candidate for being considered as independent execution fragment. By fragmenting the result space this way, a fragmentation on the input data space is induced by the structure of query expressions, rather than being defined explicitly upfront.

However, when simply using those opportunities readily available in an existing query expression it is hard to achieve a balanced query load across fragments: it is rare to find sub-expressions that are equally expensive to compute.

The structured nature of dense arrays allows the creation of new, balanced opportunities to split the query expression, such as in presence of aggregate operators. The same strategy can be used against sparse arrays. In this case however, non-uniform data distributions are most likely to induce a non-even workload distribution across the derived fragments.

Aggregate fragmentation

The following equivalence can be (repeatedly) applied to any commutative and associative aggregate:

Rule 5.17 (Aggregate fragmentation). Recall from Section 4.5.3 the definition of f_{-}.

$$\frac{\begin{array}{c} A' = \mathit{Aggregate}(f, j, A) \\ k = \text{number of fragments} \end{array}}{\begin{array}{c} A' = f_{-}(A_0, f_{-}(\cdots f_{-}(A_{k-2}, A_{k-1}) \cdots)) \\ \forall l \in [0, k[\colon A_l = \mathit{Aggregate}(f, j, \mathit{RangeSel}(A, \mathcal{O}_l, \mathcal{S}_l)) \end{array}}$$

The choice of $\mathcal{O}_l$ and $\mathcal{S}_l$ determines how the data contained into array A is fragmented across the n arrays A_l. In absence of additional information, the standard rule makes each fragment equally sized (except one when the shape of A cannot be divided equally into n fragments). □

Aggregate unfolding

Rule 5.17 can be repeatedly applied until the original aggregation function is fully *unrolled* into a series of basic binary operations For example, a sum over a vector would unroll to a series of explicit additions: $Aggregate(sum, 1, A) \mapsto A(0) + A(1) + \cdots + A(n-1)$. This transformation should be used with care, as it produces query trees whose size s is data-dependent: $s = \prod_{k=0}^{j-1} \mathcal{S}_A(k)$. Such query trees are obviously not manageable in practice for large values of s. However, when s is known to be small (where 'small' is heuristically defined), such query formulations become viable and may in some cases lead to better performance, due to the avoided cost of aggregation operations.

Aggregate unfolding on explicit index arrays. Full aggregate unfolding is particularly interesting when applied to explicit arrays. The following example illustrates a typical scenario that may benefit from such a query transformation (the example uses array syntax instead of array algebra for the sake of readability).

Example 5.11 (Aggregate unfolding on explicit index arrays).
A persistent array $A_{x,y}[100, 1000]$ and an explicit array $I[3]$ are defined:

```
A = ([100,1000],int) "A"
    X = [ 30 50 90 ]
```

We now perform an element-wise sum of the 3 vectors resulting from using X as an index vector for the first axis of A, i.e. $\sum_{i=0}^{2} A(X(i), y)$:

```
[ sum([ A(X(i),y) | i ]) | y ]
```

The summation in this expression can be unrolled to as series of additions:

```
[ A(X(0),y) + A(X(1),y) + A(X(2),y) | y ]
```

Because X is an explicit array, values $X(0)$, $X(1)$ and $X(2)$ are known at compile-time, and can be directly replaced in the original query:

```
[ A(30,y) + A(50,y) + A(90,y) | y ]
```

□

Section 6.3.2 shows the impact of aggregate unfolding on a large scale text retrieval scenario.

5.1.5 Array optimisation algorithm and cost model

Given an array expression, an array optimiser defines strategies to devise a number of equivalent formulations and select the one that is estimated to be cheaper to evaluate.

The enumeration of alternative expressions is obtained by applying transformation rules such as those described in sections 5.1.1, 5.1.3 and 5.1.2. To reduce the complexity of this enumeration, SRAM applies such rules using principles from randomised and greedy optimisation algorithms [MR96, PGL02, Joh90], as well as some heuristic strategies. A predefined (or, alternatively, randomly chosen) sequence of rules is applied to the entire expression-tree recursively (in a bottom-up or in a top-down fashion, depending on the specific rule at hand). The algorithm is greedy in that it makes locally optimal choices at each stage and never reconsiders choices that were made at previous stages. Finally, it includes some heuristics, for example in the length of the rule sequence to be applied. Also, the output of some transformation rules (e.g. those defined in Section 5.1.3) may be preferred to the original expressions due to the application of some heuristic strategies, rather than deterministic measures as cost functions (see below).

Every time a transformation rule is applied to an expression, the cost of both the old and the new expression is evaluated using a predefined cost function, and the cheapest expression retained. The cost function used in SRAM is computed on a single parameter, the total data volume processed by an operator:

$$cost(X) = size(X) + cost(children(X))$$

The simplicity of this cost function is based upon the assumption that the size of arrays in input to and in output from each operator does not depend on the physical (relational) implementation of such operator and can therefore used safely for a quick optimisation phase that precedes relational optimisation. Table 5.1 lists the instantiations of this cost function for each array algebra operator.

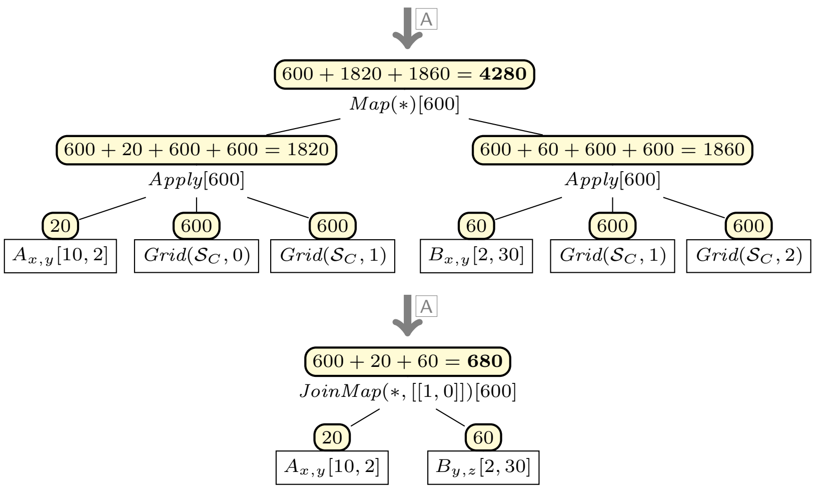
Example 5.12 (Cost function).
We consider again the transformation in Example 5.10 (*Map* to *JoinMap*), this time with real shapes, and annotate every node with cost estimations according to Table 5.1. For the sake of simplicity, we consider here dense arrays, so that costs are easier to compute from shapes. For sparse arrays, density estimations

Table 5.1: Cost measure for each array operator based on data volume

Operator	Cost
$R = \mathit{Variable}(A_{\text{def}})$	$\mathit{cost}(R) = \lvert R\rvert \cdot \delta_R$
$R = \mathit{Grid}(\mathcal{S}, j)$	$\mathit{cost}(R) = \lvert R\rvert \cdot \delta_R = \lvert R\rvert$
$R = \mathit{Const}(\mathcal{S}, c)$	$\mathit{cost}(R) = \lvert R\rvert \cdot \delta_R = 0$
$R = \mathit{Apply}(A, I_0, \ldots, I_{n-1})$	$\mathit{cost}(R) = \lvert R\rvert \cdot \delta_R + \mathit{cost}(A) + \sum_{k=0}^{n-1} \mathit{cost}(I_k)$
$R = \mathit{Map}(f, A_0, \ldots, A_{m-1})$	$\mathit{cost}(R) = \lvert R\rvert \cdot \delta_R + \sum_{k=0}^{m-1} \mathit{cost}(A_k)$
$R = \mathit{Aggregate}(f, j, A)$	$\mathit{cost}(R) = \lvert R\rvert \cdot \delta_R + \mathit{cost}(A)$
$R = \mathit{Concat}(A, B)$	$\mathit{cost}(R) = \lvert R\rvert \cdot \delta_R + \mathit{cost}(A) + \mathit{cost}(B)$
$R = \mathit{ISort}(A, \texttt{<ASC\|DESC>})$	$\mathit{cost}(R) = \lvert R\rvert \cdot \delta_R + \mathit{cost}(A) = \lvert R\rvert + \mathit{cost}(A)$
$R = \mathit{ITopN}(A, n, \texttt{<ASC\|DESC>})$	$\mathit{cost}(R) = \lvert R\rvert \cdot \delta_R + \mathit{cost}(A) = \lvert R\rvert + \mathit{cost}(A)$
$R = \mathit{Pivot}(A, \mathcal{P})$	$\mathit{cost}(R) = \lvert R\rvert \cdot \delta_R + \mathit{cost}(A)$
$R = \mathit{RangeSel}(A, \mathcal{O}, \mathcal{S})$	$\mathit{cost}(R) = \lvert R\rvert \cdot \delta_R + \mathit{cost}(A)$
$R = \mathit{Replicate}(A, k)$	$\mathit{cost}(R) = \lvert R\rvert \cdot \delta_R + \mathit{cost}(A)$
$R = \mathit{JoinMap}(f, \mathcal{J}, A_0, \ldots, A_{m-1}))$	$\mathit{cost}(R) = \lvert R\rvert \cdot \delta_R + \sum_{k=0}^{m-1} \mathit{cost}(A_k)$

must be taken into account.

```
A = ([10,2],int) "A"
B = ([2,30],int) "B"
C = [ A(x,y) * B(y,z) | x,y,z ]
```
$(\mathcal{S}_C = [10, 2, 30])$



□

5.2 Relational mapping optimisation

Both array algebra and relational algebra are suitable representations for the application of rule-based tree-rewriting optimisation techniques. Section 5.1 discusses optimisations in the array algebra domain. Section 5.3 reviews the well-known relational optimisations that are most relevant for SRAM. Important optimisation opportunities arise during the mapping between array algebra and relational algebra, by exploiting knowledge in both domains simultaneously. This is the topic of this section.

Section 5.2.1 discusses how to improve upon the straightforward relational representation scheme for sparse arrays introduced in Section 4.5.1. Section 5.2.2 introduces generic simplification mechanisms that allow for the optimisation of specific mapping rules between the two domains. Some alternative mapping rules so obtained for the *Map* are shown in Section 5.2.3.

5.2.1 Relational representation of sparse arrays

Recall from Section 4.5.1 how a sparse array A can be mapped onto a relation $\mathsf{A}(\bar{\imath}, v)$:

$$A \mapsto \mathsf{A}(\bar{\imath}, v) = \{(\bar{\imath}, A(\bar{\imath})) \mid \bar{\imath} \in \mathcal{D}_A, A(\bar{\imath}) \neq \varepsilon_A\}$$

The following sections present some optimisations over this standard representation. Although all such optimisations result in more compact representations for given array patterns, saving storage space is only a secondary goal. The main aim is to represent arrays on disk such that the reduced I/O cost involved has a beneficial impact on query processing efficiency.

Encoding sparseness

A first choice that concerns the optimisation of relational sparse array representation is whether to use a special encoding for sparse arrays at all. Such encodings imply a trade-off between the benefits they provide and the overhead they introduce inevitably (space- and/or computation-wise). For example, the storage scheme proposed in this thesis becomes less and less beneficial in terms of saved storage space as the array density grows, while the overhead introduced by sparse array evaluation remains. In such cases, it may result more efficient to pay something more in terms of storage space, storing the array as dense, and profit from a faster dense array evaluation. The decision of whether to use the sparse or dense representation for an array is an optimisation problem that can be addressed by a mixture of cost-based and heuristic-based strategies. This decision depends on many factors, including the characteristics of the particular application, the storage space available and the density of the array.

The sparse array representation can be seen as a compression mechanism. In principle, this scheme could be extended by recursively applying such a compression, on different ε values at each applications, and managing a list of missing elements (ε_A^1,ε_A^2,...) from the initial table.

Optimising special cases

In case of persistent dense arrays, all values in the primary key are present and thus form a computable sequence. As an optimisation, columns $(i_0, \ldots, i_{n-1})$ could therefore be omitted from the dense physical representation, as the full table can be regenerated by using *identity-columns* or *sequence generators*, as supported by SQL-2003. For example, given a dense array $A[3,4]$, the `SEQUENCE` construct can be used to generate a single index column for the array, as if it was one-dimensional. Finally, the one-dimensional array obtained can be re-shaped into an n-dimensional one by using the *row-major index transformation* technique described in Section 5.2.1:

```
CREATE SEQUENCE seq_int AS INTEGER START WITH 0;

CREATE VIEW A1 AS
SELECT NEXT VALUE FOR seq_int as i0, A.v as v
FROM A;

CREATE VIEW A2 AS
SELECT (i0/4)%3 as i0, i0%4 as i1, v
FROM A1;
```

A special case worth mentioning is the one of *boolean* sparse arrays. Here, the value stored in column v is always the same ($v = \neg\varepsilon$), hence for sparse boolean arrays column v can be omitted from the physical representation.

Optimising access patterns

The optimisation of different data access patterns can be achieved by means of standard relational indexing structures on top of such relations, or by explicit tuple clustering/sorting. SRAM uses the policy to store all persistent arrays in relations that are clustered on their primary key (e.g. using index-organised tables). This means that the order in which the array dimensions (that for a compound primary key) are specified matters, and puts the tuples in the underlying database relation in dimension order. Further improvements, not yet implemented in the current SRAM prototype, include array storage fragmentation into tiles (or different fragment patterns) and replication of array relations. Both tiles and array replica can be optimised independently for different access patterns.

Index transformations

The storage scheme proposed in Section 4.5.1 implies that the list of index attributes in the relational domain matches the list of axes in the array domain. However, this is not strictly necessary: one of the properties of array algebra is to be independent of its physical representation (relational, in our case).

The list of index attributes in the relational domain may be a *transformation* of the list of axes in the array domain. A useful index transformation maps every n-dimensional index $\bar{\imath}$ onto a one-dimensional index. By doing so, only one index column is needed in the relational representation, which can save much storage space and in some cases allow for a faster query execution. Because the transformation maps indices from a n-dimensional space to a one-dimensional space, the relational operations involved may need to be modified, and some of the array operations may not be easily translated into relational plans when such transformation is applied in between. For example, a simple $\mathit{Pivot}(A, 1, 0)$, which swaps the order of axes 0 and 1, may require to be executed on the n-dimensional space in the relational domain. Transformations between the two spaces may be decided at several points in the relational plan, provided that the inverse transformation is possible. We present here some of such transformation functions. Support for additional transformations, which include the Morton-order [Mor66] (also known as Z-order) and Hilbert curves [Hil91], together with other variants of space filling curves [Pea90], would be interesting future work in the context of locality-preserving transformations.

Rule 5.18 (Generalised column-major index transformation). The generalised column-major index transformation is a method for mapping multidimensional arrays indices onto the linear address space of physical memory [Knu97], used by programming languages such as Fortran and mathematical tools such as Matlab. When applied to matrices, this method has the effect of concatenating columns (see Figure 5.4a). Generalising to n dimensions, the concatenation proceeds from the last to the first dimension. Function ${}_c\Gamma_{n:1}$ takes a shape $\mathcal{S}$ and an n-dimensional index $\bar{\imath}$ as inputs, and returns the 1-dimensional index i':

$$\frac{\bar{\imath} = (i_0, \ldots, i_{n-1}) \qquad \mathcal{S} = (s_0, \ldots, s_{n-1}) \qquad i_0 < s_0, \ldots, i_{n-1} < s_{n-1}}{i' = {}_c\Gamma_{n:1}(\mathcal{S}, \bar{\imath}) = \sum_{j=0}^{n-1} \left(i_j \cdot \prod_{k=0}^{j-1} s_k \right)}$$

To map indices back onto their original n-dimensional space, the n-dimensional shape $\mathcal{S}$ is required by function ${}_c\Gamma_{1:n}$ ($\div$ and % are integer division and modulo, respectively):

$$\frac{\mathcal{S} = (s_0, \ldots, s_{n-1}) \qquad i' < \prod_{j=0}^{n-1} s_j}{\begin{array}{c} \bar{\imath} = ({}_c\Gamma_{1:n}(\mathcal{S}, i', 0), \ldots, {}_c\Gamma_{1:n}(\mathcal{S}, i', n-1)) \\ \forall j \in [1, n[\colon {}_c\Gamma_{1:n}(\mathcal{S}, i', j) = \left(i' \div \prod_{k=0}^{j-1} s_k\right) \% \, s_j \end{array}}$$

□

Rule 5.19 (Generalised row-major index transformation). The generalised row-major index transformation is a method for mapping multidimensional arrays indices onto the linear address space of physical memory [Knu97], used by programming languages such as C. Compared to the column-major transformation, the axis order is reversed. When applied to matrices, this method has the effect of concatenating rows (see Figure 5.4b). Generalising to n dimensions, the concatenation proceeds from the first to the last dimension. Function ${}_r\Gamma_{n:1}$ takes a shape $\mathcal{S}$ and an n-dimensional index $\bar{\imath}$ as inputs, and returns the 1-dimensional index i':

$$\frac{\begin{array}{c} \bar{\imath} = (i_0, \ldots, i_{n-1}) \qquad \mathcal{S} = (s_0, \ldots, s_{n-1}) \\ i_0 < s_0, \ldots, i_{n-1} < s_{n-1} \end{array}}{i' = {}_r\Gamma_{n:1}(\mathcal{S}, \bar{\imath}) = \sum_{j=0}^{n-1} \left(i_j \cdot \prod_{k=j+1}^{n-1} s_k \right)}$$

To map indices back onto their original n-dimensional space, the n-dimensional shape $\mathcal{S}$ is required by function ${}_r\Gamma_{1:n}$ ($\div$ and % are integer division and modulo, respectively):

$$\frac{\mathcal{S} = (s_0, \ldots, s_{n-1}) \qquad i' < \prod_{j=0}^{n-1} s_j}{\begin{array}{c} \bar{\imath} = ({}_r\Gamma_{1:n}(\mathcal{S}, i', 0), \ldots, {}_r\Gamma_{1:n}(\mathcal{S}, i', n-1)) \\ \forall j \in [0, n[\colon {}_r\Gamma_{1:n}(\mathcal{S}, i', j) = \left(i' \div \prod_{k=j+1}^{n-1} s_k\right) \% \, s_j \end{array}}$$

□

Column- and row-major index transformation are two variants of the same family of transformations, $\Gamma_{n:1}$ and $\Gamma_{1:n}$ (notice the absence of ${}_c$ and ${}_r$ subscripts).

The choice of which one to use depends on the specific problem at hand. It is worthwhile mentioning a particular case when they can be used together: using the complementary version of this transformation to map an array's indices back to their n-dimensional space is equivalent to reversing the array's axis order. For example, given a 2-dimensional array A:

$$\forall \bar{\imath} \in \mathcal{D}_A : {}_r\Gamma_{1:2}(\mathcal{S}_A, {}_c\Gamma_{2:1}(\mathcal{S}_A, \bar{\imath})) \equiv \mathit{Pivot}(A, [1, 0])$$

$\Gamma_{n:1}$ transformations map n-dimensional index domains to 1-dimensional index domains. This property is important for dense arrays, in that dense sequences do not require storage (see Section 5.2.1). One disadvantage of $\Gamma_{1:n}$ transformations (from 1 to n dimensions), is that they contain integer division ($\div$) and modulo (%) arithmetic. Compared to the addition ($+$) and multiplication ($\cdot$) involved in $\Gamma_{n:1}$ transformations, integer division and modulo are very slow operations, that even modern CPUs cannot optimise (for example through parallelisation). The following method modifies the original shape at hand before applying $\Gamma_{n:1}$ and $\Gamma_{1:n}$ transformations, in order to be able to perform faster division and modulo operations.

Rule 5.20 (Exploiting fast arithmetics on powers of 2). This method uses $\Gamma_{n:1}$ and $\Gamma_{1:n}$ transformations on a modified shape $\lceil\mathcal{S}\rceil^{(2)}$. In this modified shape, each dimensionality is raised up to the next number that is a power of two (e.g. $[3, 10]$ becomes $[4, 16]$):

$$\frac{\begin{array}{c}\bar{\imath} = (i_0, \ldots, i_{n-1}) \qquad \mathcal{S} = (s_0, \ldots, s_{n-1}) \\ i_0 < s_0, \ldots, i_{n-1} < s_{n-1}\end{array}}{\begin{array}{c}(\mathcal{S}', i') = \Gamma_{n:1}(\lceil\mathcal{S}\rceil^{(2)}, \bar{\imath}) \\ \lceil\mathcal{S}\rceil^{(2)} = (\lceil s_0\rceil^{(2)}, \ldots, \lceil s_{n-1}\rceil^{(2)}) \\ \lceil x\rceil^{(2)} = 2^{\lceil \log_2(x)\rceil}\end{array}}$$

The transformation that maps indices back to their n-dimensional domain uses

$\Gamma_{1:n}$, again on the modified shape $\lceil \mathcal{S} \rceil^{(2)}$:

$$\frac{\begin{array}{c} \mathcal{S} = (s_0, \ldots, s_{n-1}) \\ \lceil \mathcal{S} \rceil^{(2)} = (\lceil s_0 \rceil^{(2)} = 2^{e_0}, \ldots, \lceil s_{n-1} \rceil^{(2)} = 2^{e_{n-1}}) \\ i' < \prod_{j=0}^{n-1} \lceil s_j \rceil^{(2)} \end{array}}{\begin{array}{c} \bar{\imath} = (\Gamma_{1:n}(\lceil \mathcal{S} \rceil^{(2)}, i', 0), \ldots, \Gamma_{1:n}(\lceil \mathcal{S} \rceil^{(2)}, i', n-1)) \\ \text{with:} \\ i' \;\%\; \lceil s_k \rceil^{(2)} \equiv i' \;\texttt{\&}\; (\lceil s_k \rceil^{(2)} - 1) \\ i' \div \lceil s_k \rceil^{(2)} \equiv i' \;\texttt{>>}\; e_k \end{array}}$$

□

The advantage of using the $\lceil \mathcal{S} \rceil^{(2)}$ shape is that division and modulo operations can be performed much faster when their second operand is a power of two:

$$\text{When } a = 2^b\text{: } \begin{cases} x \;\%\; a & \equiv x \;\texttt{\&}\; (a-1) \\ x \div a & \equiv x \;\texttt{>>}\; b \end{cases}$$

Binary AND (&) and SHIFT (>>) operations can be executed very efficiently, even 10 times as fast as % and $\div$ operations. If $\lambda_{\%}$ and $\lambda_{\div}$ are the CPU-cycles saved by implementing % and $\div$ operations with & and >> operations respectively, the total number λ of CPU-cycles saved by applying a $\lceil \Gamma_{1:n} \rceil^{(2)}$ transformation instead of a $\Gamma_{1:n}$ transformation to the 1-dimensional encoding of an n-dimensional array A is:

$$\lambda = (n \cdot \lambda_{\%} + (n-1) \cdot \lambda_{\div}) \cdot |A| \cdot \delta_A$$

This cost can easily result in wall-clock time differences in the order of milliseconds for medium-complexity queries on reasonably sized arrays (e.g. the scenarios described in Chapter 6), which, depending on the application, may have a high impact on the overall performance.

By enlarging a shape $\mathcal{S}$ to $\lceil \mathcal{S} \rceil^{(2)}$, part of the associated index domain is not used by the original array (see figures 5.4c and 5.4d). This does not result in wasted storage space, as the "holes" created in the linear representation of n-dimensional correspond to non-stored tuples. However, such holes introduce additional complications, because they may be erroneously interpreted as non-stored tuples corresponding to common values in a sparse array, if not handled with care during relational processing.

Example 5.13 (Generalised row- and column- major index transformation). Consider the 3-dimensional array A with:

$$\begin{aligned}\mathcal{S}_A &= [5, 20, 12]\\ \bar{\imath} &= (3, 10, 9) \in \mathcal{D}_A\end{aligned}$$

$$\mathcal{S}'_A = [5 \cdot 20 \cdot 12] = [1200]$$

column-major index transformation:

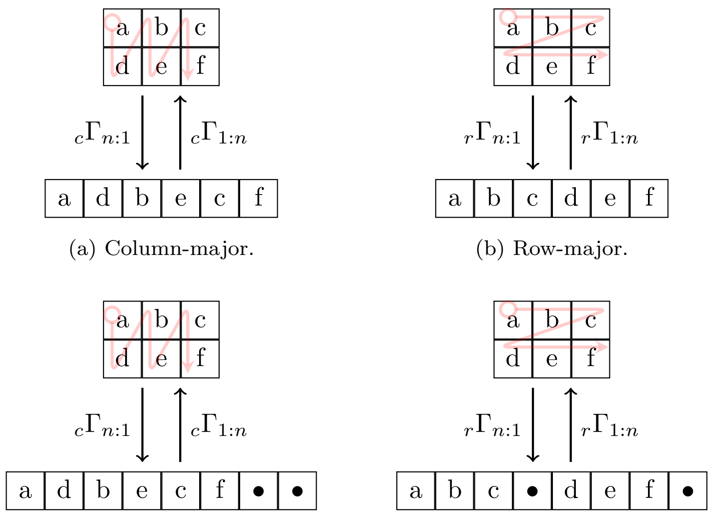
$$\begin{aligned}i' &= {}_c\Gamma_{n:1}(\mathcal{S}_A, \bar{\imath}) = 3 + 10 \cdot 5 + 9 \cdot 5 \cdot 20 = 953\\ i_0 &= {}_c\Gamma_{1:n}(\mathcal{S}_A, 953, 0) = 953 \,\%\, 5 = 3\\ i_1 &= {}_c\Gamma_{1:n}(\mathcal{S}_A, 953, 1) = (953 \div 5) \,\%\, 20 = 10\\ i_2 &= {}_c\Gamma_{1:n}(\mathcal{S}_A, 953, 2) = (953 \div (5 \cdot 20)) \,\%\, 12 = 9\end{aligned}$$

row-major index transformation:

$$\begin{aligned}i' &= {}_r\Gamma_{n:1}(\mathcal{S}_A, \bar{\imath}) = 3 \cdot 12 \cdot 20 + 10 \cdot 12 + 9 = 849\\ i_0 &= {}_r\Gamma_{1:n}(\mathcal{S}_A, 849, 0) = (849 \div (20 \cdot 12)) \,\%\, 5 = 3\\ i_1 &= {}_r\Gamma_{1:n}(\mathcal{S}_A, 849, 1) = (849 \div 12) \,\%\, 20 = 10\\ i_2 &= {}_r\Gamma_{1:n}(\mathcal{S}_A, 849, 2) = 849 \,\%\, 12 = 9\end{aligned}$$

column-major index transformation on $\lceil \mathcal{S}_A \rceil^{(2)}$:

$$\begin{aligned}\lceil \mathcal{S}_A \rceil^{(2)} &= [8, 32, 16]\\ i' &= {}_c\Gamma_{n:1}(\lceil \mathcal{S}_A \rceil^{(2)}, \bar{\imath}) = 3 + 10 \cdot 8 + 9 \cdot 8 \cdot 32 = 2387\\ i_0 &= {}_c\Gamma_{1:n}(\lceil \mathcal{S}_A \rceil^{(2)}, 2387, 0) = 2387 \,\%\, 8 = 2387 \,\&\, 7 = 3\\ i_1 &= {}_c\Gamma_{1:n}(\lceil \mathcal{S}_A \rceil^{(2)}, 2387, 1) = (2387 \div 8) \,\%\, 32 = (2387 >> 3) \,\&\, 31 = 10\\ i_2 &= {}_c\Gamma_{1:n}(\lceil \mathcal{S}_A \rceil^{(2)}, 2387, 2) = (2387 \div (8 \cdot 32)) \,\%\, 16\\ &= (2387 >> (3 + 5)) \,\&\, 15 = 9\end{aligned}$$



(a) Column-major. (b) Row-major.

(c) Column-major on $\lceil [2,3] \rceil^{(2)}$. (d) Row-major on $\lceil [2,3] \rceil^{(2)}$.

Figure 5.4: Index transformations for an array $A[2,3]$

row-major index transformation on $\lceil \mathcal{S}_A \rceil^{(2)}$:

$$
\begin{aligned}
\lceil \mathcal{S}_A \rceil^{(2)} &= [8, 32, 16] \\
i' &= {}_r\Gamma_{n:1}(\lceil \mathcal{S}_A \rceil^{(2)}, \bar{\imath}) = 3 \cdot 16 \cdot 32 + 10 \cdot 16 + 9 = 1705 \\
i_0 &= {}_r\Gamma_{1:n}(\lceil \mathcal{S}_A \rceil^{(2)}, 1705, 0) = (1705 \div (32 \cdot 16)) \,\%\, 8 = \\
&= (1705 \texttt{ >> } (5+4)) \,\texttt{\&}\, 7 = 3 \\
i_1 &= {}_r\Gamma_{1:n}(\lceil \mathcal{S}_A \rceil^{(2)}, 1705, 1) = (1705 \div 16) \,\%\, 32 = (1705 \texttt{ >> } 4) \,\texttt{\&}\, 31 = 10 \\
i_2 &= {}_r\Gamma_{1:n}(\lceil \mathcal{S}_A \rceil^{(2)}, 1705, 2) = 1705 \,\%\, 16 = 1705 \,\texttt{\&}\, 15 = 9 \qquad \square
\end{aligned}
$$

5.2.2 Arithmetic optimisation

Many expressions compute arithmetic formulae over the array data. These computations provide often the opportunity for simplification, using very basic laws of mathematics. During the translation from array algebra to relational algebra, a simple analysis takes place for every arithmetic expression. Common patterns such as

- $(x * 0) = 0, \quad (x * 1) = x$
- $(x/1) = x$
- $(x + 0) = x$
- $(x - 0) = x, \quad (x - x) = 0$
- $\log(1) = 0$
- $(x == x) = \texttt{true}$
- $(x\,! = x) = \texttt{false}$

and so on, are identified and simplified. Besides avoiding the runtime computation of such patterns, the main rationale for such a simplification is to reduce arithmetic expressions to known values, which can in turn activate other important simplification rules. In particular, arithmetic optimisation is a crucial mechanism to identify ε values in arbitrarily complex expressions and avoid their storage in intermediate results, which has several important consequences:

- Unnecessary tuples are not stored in the relational representation of sparse arrays.
- The actual density of sparse arrays stays closer to the estimated one. This makes the optimisation process more reliable.
- The complexity of generic sparse array evaluation can be drastically reduced. The effects of arithmetic optimisation can activate rules that simplify the relational expressions needed to implement sparse array computations.

A perfect example of the last item is the following rule, which has a great impact on the simplification of several mapping rules, most notably those of *Map*, *JoinMap* and *Aggregate* operators.

Rule 5.21 (Empty projection). This rule avoids the storage of relations that represent sparse arrays, when all values in their v attribute are or become equal to the default value of such arrays:

$$\frac{\begin{array}{c} Y \mapsto \mathsf{Y} = \pi_{\bar{\mathsf{I}},\mathsf{v}=f(\bar{\mathsf{v}})}(\mathsf{X}) \\ f(\mathsf{X}.\bar{\mathsf{v}}) = \varepsilon_Y \end{array}}{\mathsf{Y} \equiv \{\}}$$

□

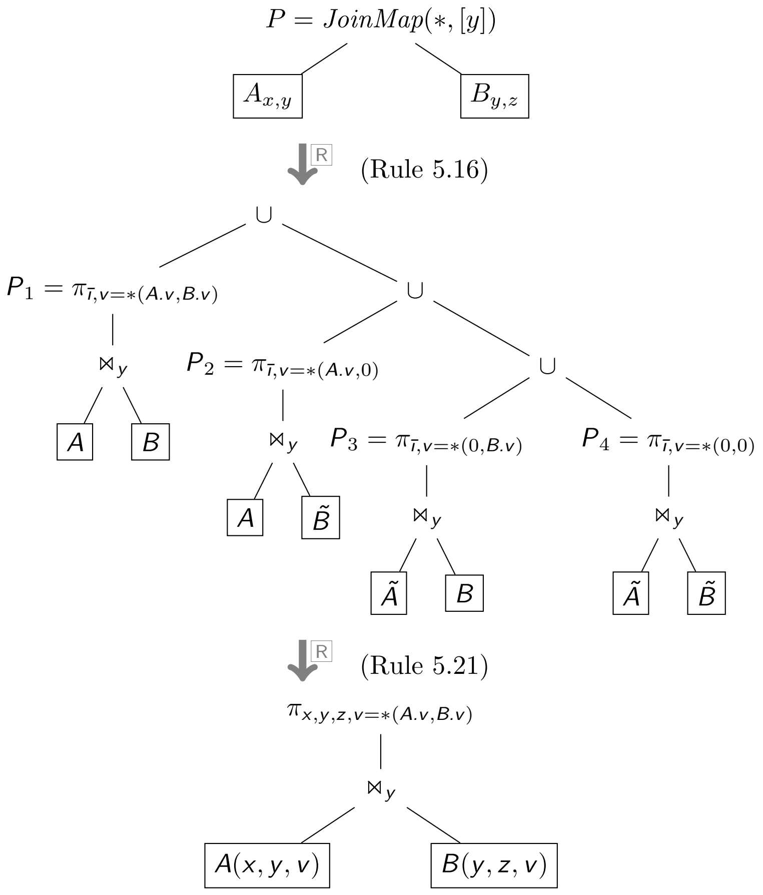
Example 5.14 (Simplifying sparse matrix multiplication).
Consider the matrix multiplication in the expression

```
C = [ sum( [ A(x,y) * B(y,z) | y ] ) | x,z ]
```

where A and B are sparse arrays with $\varepsilon_A = 0$ and $\varepsilon_B = 0$. For presentation purposes, we split the original expression in two sub-expressions: the first, `P`, considers only the multiplication part, while the second, C, performs the final summation:

```
P = [ [ A(x,y) * B(y,z) | y ] | x,z ]
C = [ sum( [ P(x,z)(y) | y ] ) | x,z ]
```

The first sub-expression defines an array P, with $\varepsilon_P = *(0,0) = 0$. The *Map* operation corresponding to function '$*$' and the associated shape-alignment procedure on arrays A and B can be substituted by a *JoinMap* operator, as in Rule 5.15. This maps onto relational domain as shown in the following steps:



The generic relational mapping of *JoinMap* between 2 arrays requires the union of 4 join operations, each representing a different portion of the result array. On top of each join, a projection computes the value of each sub-array's elements. By applying arithmetic optimisation and subsequently Rule 5.21 to such projections, 3 of them can be removed:

- $P_2.v = *(A.v, 0) = 0 \Longrightarrow P_2 \equiv \{\}$
- $P_3.v = *(0, B.v) = 0 \Longrightarrow P_3 \equiv \{\}$
- $P_4.v = *(0, 0) = 0 \Longrightarrow P_4 \equiv \{\}$

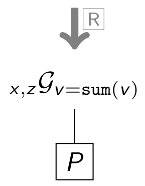
Expression P_4 is removed by Rule 5.21, as the result of $*(\varepsilon_A, \varepsilon_B)$ coincides by definition with the default value ε_P, which requires no storage. Expressions P_2 and P_3 are also removed by Rule 5.21, because $\varepsilon_A = \varepsilon_B = \varepsilon_P$. This makes the relational mapping of sparse array P simplify to sub-expression P_1, the same as if arrays A and B were dense.

The second part of the original expression computes a summation over array P:

```
C = [ sum( [ P(x,z)(y) | y ] ) | x,z ]
```

Its complete mapping onto relational algebra is as shown below:

$$\begin{tikzcd}
 & C = \mathit{Aggregate}(\mathtt{sum}, 1) \arrow[d, dash] & \\
 & \boxed{P_{y,x,z}} \arrow[d, "\boxed{\mathrm{R}}\ \text{(Rule 4.8)}"] & \\
 & \pi_{\bar{I}, v=+(G.v, *(0, M.v))} \arrow[d, dash] & \\
 & \bowtie_{\bar{I}} \arrow[dl, dash] \arrow[dr, dash] & \\
G = {}_{x,z}\mathcal{G}_{v=\mathtt{sum}(v)} \arrow[d, dash] & & M = \pi_{\bar{I}, v=((|x|\cdot|z|)-v)} \arrow[d, dash] \\
\boxed{P} & & {}_{x,z}\mathcal{G}_{v=\mathtt{count(*)}} \arrow[d, dash] \\
 & & \boxed{P}
\end{tikzcd}$$

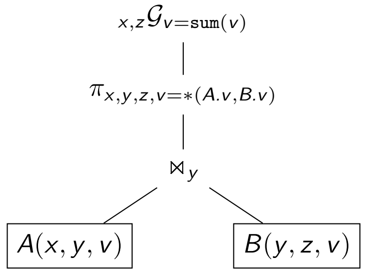


Relation G computes the summation of stored tuples in P. This is the equivalent to the mapping of array P as if it were dense. Relation M computes the number of non-stored tuples per group in relation P. The final join and projection add the contribution of non-stored tuples per group to the regular aggregation values in G. The key simplification that triggers the optimisation to the final relational tree takes place in the top projection and is triggered by arithmetic optimisation:

$$+(G.v, *(0, M.v)) = G.v \Longrightarrow \pi_{\bar{I},v=+(G.v,*(0,M.v))} = \pi_{\bar{I},v=G.v}$$

Because the value $M.v$ is no longer needed in the top expression, relation M and its join with relation G can be removed altogether.

Combining the two sub-expressions back into one, the final relational plan for the matrix multiplication between sparse arrays A and B, with $\varepsilon_A = 0$ and $\varepsilon_B = 0$, is:


$_{x,z}\mathcal{G}_{v=\mathtt{sum}(v)}$
$\pi_{x,y,z,v=*(A.v,B.v)}$
$\bowtie_y$
$A(x, y, v)$
$B(y, z, v)$

□

5.2.3 Alternative mapping rules for the *Map* operator

The *Map* operator is defined over shape-aligned arrays. This property allows to rewrite the mapping rule for *Map* from its original formulation given in Rule 4.7 to alternative ones that may result in more efficient relational query plans when some conditions holds, for example when the density of arrays in input is low, or when Rule 5.21 has a low impact. The alternative formulations considered use the anti-join and the outer-join operators.

Using the anti-join operator

Recall (4.6), the instantiation of Rule 4.7 for two sparse arrays A and B:

$$\frac{A \mapsto \mathsf{A} \qquad B \mapsto \mathsf{B} \qquad C = Map(f, A, B)}{\begin{array}{l} C \mapsto \mathsf{C} = \bigcup_{j=1}^{4} \mathsf{C}_j \\ \mathsf{C}_1 = \pi_{\bar{\mathsf{i}}, \mathsf{v}=f(\mathsf{A.v}, \mathsf{B.v})}(\mathsf{A} \bowtie_{\bar{\mathsf{i}}} \mathsf{B}) \\ \mathsf{C}_2 = \pi_{\bar{\mathsf{i}}, \mathsf{v}=f(\mathsf{A.v}, \varepsilon_B)}(\mathsf{A} \bowtie_{\bar{\mathsf{i}}} \tilde{\mathsf{B}}) \\ \mathsf{C}_3 = \pi_{\bar{\mathsf{i}}, \mathsf{v}=f(\varepsilon_A, \mathsf{B.v})}(\tilde{\mathsf{A}} \bowtie_{\bar{\mathsf{i}}} \mathsf{B}) \\ \mathsf{C}_4 = \pi_{\bar{\mathsf{i}}, \mathsf{v}=f(\varepsilon_A, \varepsilon_B)}(\tilde{\mathsf{A}} \bowtie_{\bar{\mathsf{i}}} \tilde{\mathsf{B}}) \equiv \{\} \end{array}}$$

Rule 5.21 removes expression C_4, as its element values are by definition equal to ε_C. Expressions C_2 and C_3 still contain $\tilde{\mathsf{A}}$ and $\tilde{\mathsf{B}}$ relations, that may be very large when the density of arrays A and B is small, causing severe efficiency issues.

Because A and B are shape-aligned, relations A, B, $\tilde{\mathsf{A}}$ and $\tilde{\mathsf{B}}$ all share the same schema and the same compound primary key $\bar{\mathsf{i}}$. This, together with the fact that all join conditions are on the primary key $\bar{\mathsf{i}}$, makes the following rule applicable:

Rule 5.22 (Join to anti-join for retrieving non-stored indices). When A and B are shape-aligned, i.e. when $\mathcal{S}_A = \mathcal{S}_B$, the primary key join $\mathsf{A} \bowtie_{\bar{\mathsf{i}}} \tilde{\mathsf{B}}$ matches all the indices stored in A and in $\tilde{\mathsf{B}}$. This is equivalent to matching all the indices stored in A but not stored in B. The latter formulation allows to avoid computation of relation $\tilde{\mathsf{B}}$ by means of an anti-join between stored relations A and B:

$$\frac{A \mapsto \mathsf{A} \qquad B \mapsto \mathsf{B} \qquad \mathcal{S}_A = \mathcal{S}_B}{\mathsf{A} \bowtie_{\bar{\mathsf{i}}} \tilde{\mathsf{B}} \equiv \mathsf{A} \rhd_{\bar{\mathsf{i}}} \mathsf{B}}$$

Depending on the density of array B, therefore on the relative sizes of relations B and $\tilde{\mathsf{B}}$, the anti-join operation may be cheaper to evaluate than the original join operation. Although the latter requires in principle a simpler evaluation, its cost may be dominated by the materialisation of relation $\tilde{\mathsf{B}}$, potentially very large. □

By applying Rule 5.22 to 4.6, an equivalent formulation that uses the anti-join operator can be derived for the *Map* over two sparse aligned arrays A and B:

$$\frac{\begin{array}{l} A \mapsto \mathsf{A} \qquad B \mapsto \mathsf{B} \\ C = \mathit{Map}(f, A, B) \end{array}}{\begin{array}{l} C \mapsto \mathsf{C} = \bigcup_{j=1}^{4} \mathsf{C}_j \\ \mathsf{C}_1 = \pi_{\check{I}, v=f(\mathsf{A}.v, \mathsf{B}.v)}(\mathsf{A} \bowtie_{\check{I}} \mathsf{B}) \\ \mathsf{C}_2 = \pi_{\check{I}, v=f(\mathsf{A}.v, \varepsilon_B)}(\mathsf{A} \rhd_{\check{I}} \mathsf{B}) \\ \mathsf{C}_3 = \pi_{\check{I}, v=f(\varepsilon_A, \mathsf{B}.v)}(\mathsf{B} \rhd_{\check{I}} \mathsf{A}) \\ \mathsf{C}_4 = \pi_{\check{I}, v=f(\varepsilon_A, \varepsilon_B)}(\tilde{\mathsf{A}} \bowtie_{\check{I}} \tilde{\mathsf{B}}) \equiv \{\} \end{array}} \tag{5.4}$$

Expression C_4 is always removed by Rule 5.21. The join sequences of the different parts of relation C should read informally as:

[C_1: $\mathsf{A} \bowtie_{\check{I}} \mathsf{B}$] All indices stored in A and B.

[C_2: $\mathsf{A} \rhd_{\check{I}} \mathsf{B}$] All indices stored in A but not in B.

[C_3: $\mathsf{B} \rhd_{\check{I}} \mathsf{A}$] All indices stored in B but not in A.

The same formulation for three sparse arrays A, B and C is shown in (5.5):

$$\frac{\begin{array}{l} A \mapsto \mathsf{A} \qquad B \mapsto \mathsf{B} \qquad C \mapsto \mathsf{C} \\ D = \mathit{Map}(f, A, B, C) \end{array}}{\begin{array}{l} D \mapsto \mathsf{D} = \bigcup_{j=1}^{8} \mathsf{D}_j \\ \mathsf{D}_1 = \pi_{\check{I}, v=f(\mathsf{A}.v, \mathsf{B}.v, \mathsf{C}.v)}(\mathsf{A} \bowtie_{\check{I}} \mathsf{B} \bowtie_{\check{I}} \mathsf{C}) \\ \mathsf{D}_2 = \pi_{\check{I}, v=f(\mathsf{A}.v, \mathsf{B}.v, \varepsilon_C)}(\mathsf{A} \bowtie_{\check{I}} \mathsf{B} \rhd_{\check{I}} \mathsf{C}) \\ \mathsf{D}_3 = \pi_{\check{I}, v=f(\mathsf{A}.v, \varepsilon_B, \mathsf{C}.v)}(\mathsf{A} \bowtie_{\check{I}} \mathsf{C} \rhd_{\check{I}} \mathsf{B}) \\ \mathsf{D}_4 = \pi_{\check{I}, v=f(\varepsilon_A, \mathsf{B}.v, \mathsf{C}.v)}(\mathsf{B} \bowtie_{\check{I}} \mathsf{C} \rhd_{\check{I}} \mathsf{A}) \\ \mathsf{D}_5 = \pi_{\check{I}, v=f(\mathsf{A}.v, \varepsilon_B, \varepsilon_C)}(\mathsf{A} \rhd_{\check{I}} \mathsf{B} \rhd_{\check{I}} \mathsf{C}) \\ \mathsf{D}_6 = \pi_{\check{I}, v=f(\varepsilon_A, \mathsf{B}.v, \varepsilon_C)}(\mathsf{B} \rhd_{\check{I}} \mathsf{A} \rhd_{\check{I}} \mathsf{C}) \\ \mathsf{D}_7 = \pi_{\check{I}, v=f(\varepsilon_A, \varepsilon_B, \mathsf{C}.v)}(\mathsf{C} \rhd_{\check{I}} \mathsf{A} \rhd_{\check{I}} \mathsf{B}) \\ \mathsf{D}_8 = \pi_{\check{I}, v=f(\varepsilon_A, \varepsilon_B, \varepsilon_C)}(\tilde{\mathsf{A}} \bowtie_{\check{I}} \tilde{\mathsf{B}} \bowtie_{\check{I}} \tilde{\mathsf{C}}) \equiv \{\} \end{array}} \tag{5.5}$$

The join sequences of the different parts of relation D should read informally as:

[D_1: $\mathsf{A} \bowtie_{\check{I}} \mathsf{B} \bowtie_{\check{I}} \mathsf{C}$] All indices stored in A, B and C.

[D_2: $\mathsf{A} \bowtie_{\check{I}} \mathsf{B} \rhd_{\check{I}} \mathsf{C}$] All indices stored in A and B but not in C.

[D_3: $\mathsf{A} \bowtie_{\check{I}} \mathsf{C} \rhd_{\check{I}} \mathsf{B}$] All indices stored in A and C but not in B.

[D_4: $B \bowtie_{\bar{I}} C \rhd_{\bar{I}} A$] All indices stored in B and C but not in A.

[D_5: $A \rhd_{\bar{I}} B \rhd_{\bar{I}} C$] All indices stored in A but not in B nor in C.

[D_6: $B \rhd_{\bar{I}} A \rhd_{\bar{I}} C$] All indices stored in B but not in A nor in C.

[D_7: $C \rhd_{\bar{I}} A \rhd_{\bar{I}} B$] All indices stored in C but not in A nor in B.

Generalisation of (5.4) and (5.5) to m arrays in input is shown in Rule 5.23:

Rule 5.23 (Relational mapping for *Map* using anti-join join).

$$\frac{\begin{array}{l}\forall l \in [0; m[: A_l \mapsto A_l \\ B = Map(f, A_0, \ldots, A_{m-1})\end{array}}{\begin{array}{l}B \mapsto B = \bigcup_{j=1}^{2^m} B_j \\ B_1 = \pi_{\bar{I}, v=f(A_0.v, \ldots, A_{m-1}.v)}(A_0 \bowtie_{\bar{I}} \cdots \bowtie_{\bar{I}} A_{m-1}) \\ B_2 = \pi_{\bar{I}, v=f(A_0.v, \ldots, A_{m-2}.v, \varepsilon_{A_{m-1}})}(A_0 \bowtie_{\bar{I}} \cdots \bowtie_{\bar{I}} A_{m-2} \rhd_{\bar{I}} A_{m-1}) \\ \vdots \\ B_{2^m-1} = \pi_{\bar{I}, v=f(\varepsilon_{A_0}, \ldots, \varepsilon_{A_{m-2}}, A_{m-1}.v)}(A_{m-1} \rhd_{\bar{I}} \cdots \rhd_{\bar{I}} A_1 \rhd_{\bar{I}} A_0) \\ B_{2^m} = \pi_{\bar{I}, v=f(\varepsilon_{A_0}, \ldots, \varepsilon_{A_{m-1}})}(\tilde{A}_0 \bowtie_{\bar{I}} \cdots \bowtie_{\bar{I}} \tilde{A}_{m-1})\end{array}}$$

□

Using the outer-join operator

Recall from previous section (5.4), the formulation of *Map* for two arrays in input, that uses anti-join operators:

$$\frac{\begin{array}{c}A \mapsto A \qquad B \mapsto B \\ C = Map(f, A, B)\end{array}}{\begin{array}{l}C \mapsto C = \bigcup_{j=1}^{4} C_j \\ C_1 = \pi_{\bar{I}, v=f(A.v, B.v)}(A \bowtie_{\bar{I}} B) \\ C_2 = \pi_{\bar{I}, v=f(A.v, \varepsilon_B)}(A \rhd_{\bar{I}} B) \\ C_3 = \pi_{\bar{I}, v=f(\varepsilon_A, B.v)}(B \rhd_{\bar{I}} A) \\ C_4 = \pi_{\bar{I}, v=f(\varepsilon_A, \varepsilon_B)}(\tilde{A} \bowtie_{\bar{I}} \tilde{B}) \equiv \{\}\end{array}}$$

Because A and B share the same relational schema and all the join and anti-join conditions are defined on their primary keys, we are allowed to rewrite such a formulation by means of a single outer-join operation:

$$\frac{A \mapsto A \qquad B \mapsto B \qquad C = Map(f, A, B)}{\begin{array}{l} C \mapsto C = \pi_{\bar{\imath}, v = f(A.v \vee \varepsilon_A, B.v \vee \varepsilon_B)}(A \mathbin{=\check{\bowtie}=_{\bar{\imath}}} B) \\ (A \mathbin{=\check{\bowtie}=_{\bar{\imath}}} B) = \pi_{(\forall i \in \bar{\imath}:\ A.i \vee B.i), \bar{v}}(A \mathbin{=\bowtie=_{\bar{\imath}}} B) \end{array}} \tag{5.6}$$

This captures with one single relational operator all four cases described in (4.6) and (5.4). Expression $(A \mathbin{=\check{\bowtie}=_{\bar{\imath}}} B)$ makes sure that no `NULL` value survives the outer-join operation. Either side of the outer-join may produce `NULL` values for index attributes, but not both sides at the same time. This means that the correct values for index attributes are to be found on either side. Projection $\pi_{\forall i \in \bar{\imath}:\ A.i \vee B.i}$ propagates the one that is not equal to `NULL` between $A.\bar{\imath}$ and $B.\bar{\imath}$.

Removal of C_2 and/or C_3 in (4.6), and consequently in (5.4), corresponds to substitution of the outer-join in (5.6) for left join or inner join operations respectively. Table 5.2 describes this informally, i.e. ignoring projections.

Table 5.2: Effect of Rule 4.7 on (5.4) and (5.6)

A, B	(5.4)	(5.6)
$\exists\varepsilon_A, \exists\varepsilon_B$	$C = (C_1 \cup C_2 \cup C_3)$	$C = (A \mathbin{=\check{\bowtie}=_{\bar{\imath}}} B)$
$\nexists\varepsilon_A, \exists\varepsilon_B$	$C = (C_1 \cup C_2)$	$C = (A \mathbin{\overset{\leftarrow}{=\bowtie}_{\bar{\imath}}} B)$
$\exists\varepsilon_A, \nexists\varepsilon_B$	$C = (C_1 \cup C_3)$	$C = (B \mathbin{\overset{\leftarrow}{=\bowtie}_{\bar{\imath}}} A)$
$\exists\varepsilon_A, \nexists\varepsilon_B$ $f(\varepsilon_A, B.v) = \varepsilon_C$	$C = C_1$	$C = (A \bowtie_{\bar{\imath}} B)$
$\nexists\varepsilon_A, \nexists\varepsilon_B$	$C = C_1$	$C = (A \bowtie_{\bar{\imath}} B)$

Likewise, (5.5) for three sparse arrays A, B and C can be reformulated in terms of a single outer-join operation in replacement for a union of anti-join operations, as shown in (5.7):

$$\frac{A \mapsto A \qquad B \mapsto B \qquad C \mapsto C \qquad D = Map(f, A, B, C)}{\begin{array}{l} D \mapsto D = \pi_{\bar{\imath}, v = f(A.v \vee \varepsilon_A, B.v \vee \varepsilon_B, C.v \vee \varepsilon_C)}(A \mathbin{=\check{\bowtie}=_{\bar{\imath}}} B \mathbin{=\check{\bowtie}=_{\bar{\imath}}} C) \\ (A \mathbin{=\check{\bowtie}=_{\bar{\imath}}} B) = \pi_{(\forall i \in \bar{\imath}:\ A.i \vee B.i), \bar{v}}(A \mathbin{=\bowtie=_{\bar{\imath}}} B) \end{array}} \tag{5.7}$$

The removal of sub-expressions from (5.5) by the application of Rule 5.21 corresponds to the reduction of some full outer-joins to left-outer and inner joins in (5.7). Table 5.3 describes this informally, i.e. ignoring projections. A comparison between the second and the third rows of Table 5.3 shows why the solution based on outer-join is not always to be preferred to the one based on anti-joins. Outer-join operators can capture the result of unions of joins and anti-joins with single operations, which is potentially more efficient. However, they can only express computations with larger granularity. The example shows that both expressions $D = (D_1 \cup D_2 \cup D_4 \cup D_6)$ and $D = (D_1 \cup D_6)$ in (5.5) correspond to the same expression $D = (B \overleftarrow{=\!\bowtie}_{\bar{\imath}} A \overleftarrow{=\!\bowtie}_{\bar{\imath}} C)$ in (5.7). Expression $(R \overleftarrow{=\!\bowtie}_{\bar{\imath}} S)$ is defined as a simple projection on top of standard left-outer-join: $\pi_{R.\bar{\imath},\bar{v}}(R =\!\bowtie_{\bar{\imath}} S)$

The choice of the most efficient solution in each case can only be made by estimating the cost of each query plan in a relational optimiser.

Table 5.3: Effect of Rule 4.7 on (5.5) and (5.7) (few cases)

A, B, C	(5.5)	(5.7)
$\exists\varepsilon_A, \exists\varepsilon_B, \exists\varepsilon_C$	$D = \bigcup_{j=1}^{8} D_j$	$D = (A =\!\check{\bowtie}\!=_{\bar{\imath}} B =\!\check{\bowtie}\!=_{\bar{\imath}} C)$
$\exists\varepsilon_A, \nexists\varepsilon_B, \exists\varepsilon_C$	$D = (D_1 \cup D_2 \cup D_4 \cup D_6)$	$D = (B \overleftarrow{=\!\bowtie}_{\bar{\imath}} A \overleftarrow{=\!\bowtie}_{\bar{\imath}} C)$
$\exists\varepsilon_A, \nexists\varepsilon_B, \exists\varepsilon_C$ $f(\varepsilon_A, B.v, C.v) = \varepsilon_D$	$D = (D_1 \cup D_6)$	$D = (B \overleftarrow{=\!\bowtie}_{\bar{\imath}} A \overleftarrow{=\!\bowtie}_{\bar{\imath}} C)$
$\nexists\varepsilon_A, \exists\varepsilon_B, \nexists\varepsilon_C$	$D = (D_1 \cup D_3)$	$D = (A \bowtie_{\bar{\imath}} C \overleftarrow{=\!\bowtie}_{\bar{\imath}} B)$
$\nexists\varepsilon_A, \nexists\varepsilon_B, \nexists\varepsilon_C$	$D = D_1$	$D = (A \bowtie_{\bar{\imath}} B \bowtie_{\bar{\imath}} C)$

The generalisation of (5.6) and (5.7) to m arrays in input is shown in Rule 5.24:

Rule 5.24 (Relational mapping for *Map* using outer-join).

$$\frac{\begin{array}{l}\forall l \in [0; m[\colon A_l \mapsto A_l \\ B = Map(f, A_0, \dots, A_{m-1})\end{array}}{\begin{array}{l}B \mapsto B = \pi_{\check{\imath}, v = f(A_0.v \vee \varepsilon_{A_0}, \dots, A_{m-1}.v \vee \varepsilon_{A_{m-1}})}(X) \\ X = A_0 =\!\check{\bowtie}\!=_{\bar{\imath}} \cdots =\!\check{\bowtie}\!=_{\bar{\imath}} A_{m-1} \\ (R =\!\check{\bowtie}\!=_{\bar{\imath}} S) = \pi_{(\forall i \in \bar{\imath}\colon R.i \vee S.i), \bar{v}}(R =\!\bowtie\!=_{\bar{\imath}} S)\end{array}}$$

□

By applying Rule 5.21, Rule 5.24 can be reduced to a more efficient variant that uses a combination of outer and inner joins. This is possible by splitting the set $\{A_0, \dots, A_{m-1}\}$ in two disjunct sets $\{A_0^-, \dots, A_{q-1}^-\}$ and $\{A_0^+, \dots, A_{w-1}^+\}$, and reformulating the union of the 2^m cases in Rule 5.23 into "the computation of all indices stored in all the relations $A_0^-, \dots, A_{q-1}^-$ but not stored in any of the relations $A_0^+, \ldots, A_{w-1}^+$":

Rule 5.25 (Relational mapping for *Map* using outer and inner joins).

$$\frac{\begin{array}{l}\forall l \in [0; m[: A_l \mapsto A_l \\ B = \mathit{Map}(f, A_0, \ldots, A_{m-1})\end{array}}{\begin{array}{l} B \mapsto B = \pi_{\check{\imath}, v = f(A_0.v \vee \varepsilon_{A_0}, \ldots, A_{m-1}.v \vee \varepsilon_{A_{m-1}})}(X) \\ X = A_0^- \bowtie_{\bar{\imath}} \cdots \bowtie_{\bar{\imath}} A_{q-1}^- \overleftarrow{=\!\bowtie}_{\bar{\imath}} A_0^+ \overleftarrow{=\!\bowtie}_{\bar{\imath}} \cdots \overleftarrow{=\!\bowtie}_{\bar{\imath}} A_{w-1}^+ \\ (R \overleftarrow{=\!\bowtie}_{\bar{\imath}} S) = \pi_{R.\bar{\imath}, \bar{v}}(R =\!\bowtie_{\bar{\imath}} S) \end{array}}$$

□

5.3 Relational optimisation

Relational optimisation is known to be a NP-complete problem [IK84, SM97]. This forces the optimisation process to limit the space of query plan alternatives to be searched, for example by exploring only a subset of the possible transformation rules. For this reason, most available relational optimisers come with a set of transformation rules that is most likely to be suited to the typical query scenarios of the expected application domain. Because the development of relational technology was tightly coupled to the business-oriented application domain, supported by the SQL query language, most available relational software, including optimisers, are geared towards business-oriented application needs. This is the main reason why, although possible, translating array expressions to SQL is not an optimal choice: the array query language and SQL are designed to address different application domains and therefore they need to exploit different handles in order to use relational technology efficiently.

The relational layer of SRAM implements standard relational algebra. However, the mapping phase between the array and the relational domains, described in Section 5.2, translates additional array-specific knowledge and makes it available to the relational optimiser, which is then able to rely on more focused pieces of information and use more effectively the set of transformation rules that are expected to have a high impact.

5.3.1 Domain-independent optimisation handles

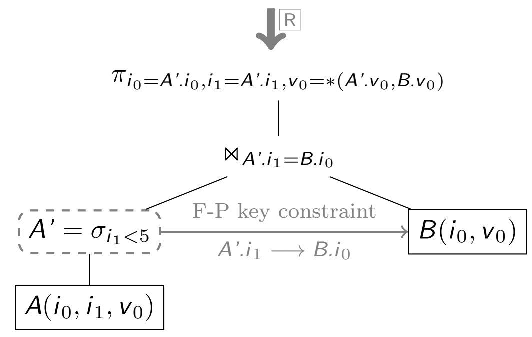
The following is a sample of properties and pieces of information that may be difficult to exploit by an available (SQL-oriented) relational system that is given an expression coming from the array domain:

- all relations have a very strict schema: $n-1$ index columns, one or more value columns;
- the domain of index values is known;
- index columns of some relations (those resulting from *IndexTable* and dense arrays) are organised as predictable dense sequences;
- index columns together always form a compound primary key, also for intermediate relations.

The following example shows how some integrity constraints can be derived due to the characteristics of the involved relations.

Example 5.15 (Derived referential constraints).
Consider the following expression between a sparse array $A[10,8]$ and a dense array $B[5]$:

```
A = ([10,8],int) sparse(0,0.1) "A"
B = ([5],int)    dense         "B"
[ A(x,y) * B(y) | x, y<5]
```



The following integrity constraints exist:

- $(A.i_0, A.i_1) \longrightarrow A$: $(A.i_0, A.i_1)$ is a primary key for A
- $B.i_0 \longrightarrow B$: $B.i_0$ is a primary key for B.

No foreign key can be defined on relation A to relation B. However, after the selection on relation A, the domain of its i_1 axis becomes identical to the domain of attribute $B.i_0$. Therefore, a query-driven referential constraint can be derived. If we name $A'=\sigma_{i_1<5}(A)$:

- $A'.i_1 \longrightarrow B.i_0$: $A'.i_1$ is a foreign key for the primary key $B.i_0$.

Such a referential constraint can be inferred without any array-specific knowledge, but with the availability of the following pieces of information: *(i)* values in $B.i_0$ form a dense sequence of integer numbers from o to 4; *(ii)* values in $A'.i_1$ range from 0 to 4. Note that the latter requires a static analysis of the selection criterion. □

The referential constraint inferred in Example 5.15 is not likely to be inferred by many available relational optimisers. This is not due to complexity reasons, but rather to its expected impact: while that pattern occurs very often for relations representing arrays, it may be considered unimportant by the developers of a general-purpose database system.

5.3.2 Standard transformation rules with high impact

Standard relational optimisers may include rules that are not necessary when dealing with such well-structured relations and a restricted set of operations (e.g. only equi-join, only selections on index attributes). On the contrary, other rules prove to be highly effective in the array scenario.

Outer-join simplification

Outer-join is an expensive operator and should be simplified to inner-join whenever it can be proved that such a simplification does not alter the final result. This can be informally described as follows: if all the NULL values produced by an outer-join operator are discarded by a following operator before they can influence the final result, then they could as well not be produced. Therefore, the outer-join originating them can be turned into an inner-join. When only the NULL values originating from either the left or right side of an outer-join are to be discarded, this can be turned into a left- or right-outer-join respectively. As a simple example, consider the following relational expression:

$$\sigma_{A.a>10}(A =\bowtie=_{A.a=B.b} B)$$

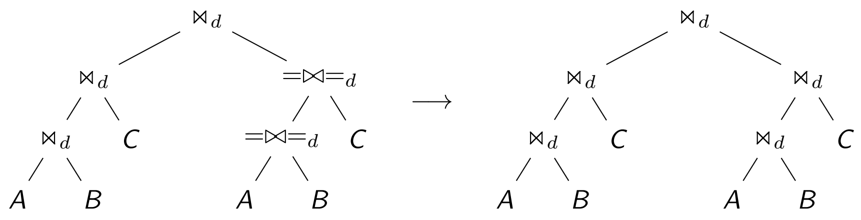
The implicit selection of non-NULL values for $A.a$ allows to rewrite the full outer-join into a left outer-join, which avoids creation of those NULL values in the first place:

$$\sigma_{A.a>10}(A =\bowtie_{A.a=B.b} B)$$

Less trivial patterns may discard NULL values, for example, by means of a subsequent inner-join with a relation that is guaranteed not to contain NULL values in the join attribute (e.g. when the attribute is a primary key). A full description of this family of relational optimisation rules can be found in [GLR97].

The application of outer-join simplification in the array domain arises from a previously described optimisation step. Rules 5.24 and 5.25 are alternative translation rules for the *Map* operator, aimed at better exploitation of shape alignment and less data management overhead. To this end, the optimiser introduces outer-join operators to replace unions of inner-join and anti-join operators, possibly more expensive. While such newly introduced outer-join operators may optimise the sub-query they belong to, it may happen that they perform more work than necessary in the context of a larger query plan.

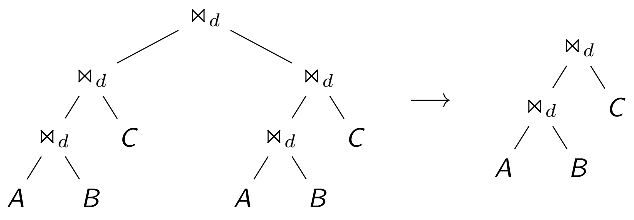
Section 6.3.2 describes in details a real scenario, a large-scale text retrieval application powered by SRAM, where outer-join simplification plays an important role. An array expression is composed by two main parts, that we report here graphically (each relation contains an attribute d which is primary key):



Because the top-level join is performed on a primary key (thus not-`NULL`), all outer-join operations can be turned into inner-join operations (see the complete discussion in Section 6.3.2 for further details).

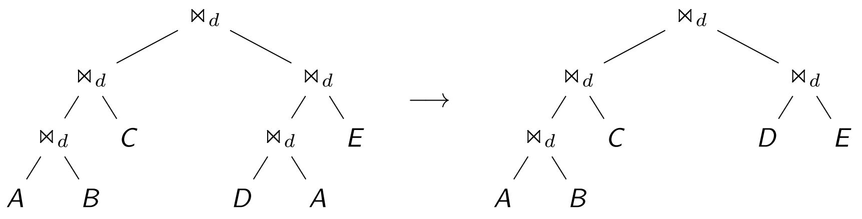
Self-join elimination

Joining a relation with itself, with an equality join condition on the relation's primary key is a superfluous operation that can be simplified away. This query pattern is not as unusual as it may seem, as it can result from rather common array expressions like `[ A(x) * A(x) | x ]`. It can also arise from previous optimisations steps. For example, the result of the outer-join simplification example shown above is a primary key join between two identical sub-trees:



Redundant join elimination

Self-join elimination can be seen as an extreme case of elimination of redundant join operations. Consider the following transformation:



The conditions that allows to remove relation A from the second sub-tree are: *(i)* the same relation appears in both the left and the right sub-trees of a primary-key inner join; *(ii)* the relation is propagated up to the main inner join by means of primary-key inner joins; *(ii)* no operation in between modifies the attributes that are used in join conditions (e.g. projections). In essence, two identical relations may appear in a query tree (even in different branches and at different depths) and be simplified by removing one of the two when the query tree transports untouched all their tuples up to the result.

Join reordering

Because the inner join operator is commutative (and associative under certain conditions), the order in which a relational system joins tables does not change the final result-set of the query. However, join order may have an enormous impact on the cost of the complete query, so choosing the best join order becomes very important. Here, we use again the result of a previous optimisation step (self-join elimination), to show how this can be further refined, computing most selective joins first:

$$\begin{tikzcd}
 & & \bowtie_d \arrow[dl, dash] \arrow[dr, dash] & & & & \bowtie_d \arrow[dl, dash] \arrow[dr, dash] & \\
 & \bowtie_d \arrow[dl, dash] \arrow[dr, dash] & & C & \longrightarrow & \bowtie_d \arrow[dl, dash] \arrow[dr, dash] & & A \\
A & & B & & B & & C &
\end{tikzcd}$$

5.4 Physical optimisation: relational back-end

Many attempts of using database technology as a back-end for scientific applications have failed in the past, due to the lack of runtime performance that such solutions provided in comparison with ad-hoc data management solutions. For example, Howe and Maier [HM04] were successful in applying the array datamodel to a scientific dataset, using real examples from an Environmental Observation and Forecasting System. However, despite the power of their algebraic framework, efficient data access was achieved by a custom-built data management layer, after the database approach had been proven too slow for their needs. In the Information Retrieval field, the first attempts of an integration with database solutions date back to the seventies (e.g. [Mac79, Cra81]). Until today, no further attempt has succeeded [CRW05, Wei07], consolidating the belief that the benefits of database technology cannot be used to without a significant loss in term of performance.

The statement "databases cannot support efficiently data-intensive applications" (such as IR) proved to be true with all the existing database engines that were developed following the classical design principles that stem from general purpose OLTP roots and historical hardware architectures. These design principles in a nutshell focus on the fact that traditionally disk I/O is the main bottleneck, while CPU processing is fast. Because query execution is expected to be I/O bound, no or little attention is paid to exploiting the processing power of the CPU. This picture has changed dramatically with modern storage hardware and when data is physically structured in such a way that sequential access to disk is predominant. In such cases, I/O throughput can be so high that even modern CPUs cannot keep up with it. The design of next-generation database engines has started from dropping the assumption that data cannot be delivered at high rates to the CPU. This immediately calls for completely new query execution strategies and algorithms, that finally focus on the exploitation of modern CPUs capabilities (e.g. large high-speed cache memories, super-scalar design, branch prediction, speculative execution, out-of-order execution), enabling data-crunching rates that can compete with those of customised solutions.

5.4.1 Using the **MonetDB/X100** backend

This section gives a brief introduction on the experimental MonetDB/X100 database engine [1], which was identified as the next-generation database engine to best fit in the scope of the SRAM project (other column-store, CPU-efficient engines, such as MonetDB [CWI], would be possible candidates).

MonetDB/X100 provides a non-declarative relational query language, and does not include an automatic query optimiser. SRAM, when used in combination with MonetDB/X100, maps relational algebra expressions directly onto the MonetDB/X100 query language, using its ad-hoc relational optimiser, which adopts common strategies, such as join reordering and join elimination (see Section 5.3). Chapter 6 details on the evaluation of experimental results using SRAM in combination with MonetDB/X100.

MonetDB/X100 overview

MonetDB/X100 is an experimental relational database engine, optimised for high performance data warehousing and OLAP workloads. It relies on the concept of *vectorised in-cache* query execution to achieve good CPU utilisation [BZN05], and a column-oriented storage manager that provides transparent *light-weight data compression* [ZHNB06] to improve I/O-bandwidth utilisation. An overview of the system architecture is presented in Figure 5.5.

MonetDB/X100 offers a language based on physical relational algebra, providing operators such as e.g. `Scan, ScanSelect, Project, Aggr, TopN, Sort, MergeJoin, HashJoin`.

Vectorised in-cache query execution.

Figure 5.5 shows an operator tree, being evaluated within MonetDB/X100 in a pipelined fashion, using the traditional *open()*, *next()*, *close()* interface from the Volcano [Gra94] iterator model. However, each *next()* call within MonetDB/X100 does not return a single tuple, as is the case in most traditional DBMSs, but a collection of *vectors*, with each vector containing a small horizontal slice of a single column. Vectorisation of the iterator pipeline allows MonetDB/X100 *primitives*, which are responsible for computing core functionality such as addition and multiplication, to be implemented as simple loops over vectors. This results in function call overheads being amortised over a full vector of values instead of a single tuple, and allows the compiler to produce data-parallel code that can be

[1]As of July 29th, 2009, the MonetDB/X100 engine development continues as the Ingres/Vectorwise project [Vec].

executed efficiently on modern CPUs. Furthermore, the size of a vector is chosen in such a way that all vectors needed by a query fit the CPU cache. This avoids materialisation of tuples that are being passed from one operator to the next, minimising main memory access overheads. Such a *vectorised in-cache* architecture allows MonetDB/X100 query evaluation to be an order of magnitude faster than existing technology on data- and query-intensive workloads [BZN05].

Column-oriented storage and compression

The processing power of MonetDB/X100 can make the system extremely I/O-hungry on certain queries. If the database does not fit in main memory, the only solution to this problem is to increase the available I/O bandwidth. This can be done by adding more hardware, or by optimising the DBMSs buffer manager for bandwidth utilisation. With respect to the latter, MonetDB/X100 employs a buffer manager, called ColumnBM, that relies on a column-oriented storage scheme, to avoid reading unnecessary columns from disk. Further, the granularity of disk accesses is in blocks of several megabytes, to optimise for fast sequential I/O.

MonetDB/X100 takes the point of I/O-bandwidth utilisation even further, by integrating ultra light-weight column compression, that happens on the boundary between RAM and CPU cache (whereas other compressed database systems typically target the disk-RAM boundary). These compression schemes are integrated into the DBMS in such a way that data blocks are stored in compressed form in memory (such that more data fits the buffer cache), and data is only decompressed on-demand, at vector granularity, directly into the CPU cache, where it is fed into the operator pipeline, without writing the uncompressed data back to main memory, as can be seen in Figure 5.5.

For compression to improve speed when using RAID storage systems delivering hundreds of megabytes per second, we need decompression routines that can uncompress several gigabytes of data per second. To reach such speeds, MonetDB/X100 is provided with the compression algorithms PFOR and PFOR-DELTA [ZHNB06], that are designed to sacrifice some performance in terms of compression ratio, in exchange for fast decompressibility.

Frame Of Reference (FOR) is a database compression method that stores numerical table columns in a disk-block as the increment to a certain base value. The increments are represented in a small integer, with a fixed bit width b. Each disk-block may define a different base value and b. PFOR (Patched FOR) is an extension of FOR [ZHNB06] that stores values as either a *code* or an *exception*. In PFOR, codes are small integer increments to a base, like in FOR. Exception values

	PFOR-DELTA			**carryover-12**		
	comp ratio	comp MB/s	dec MB/s	comp ratio	comp MB/s	dec MB/s
INEX	1.75	679	3053	2.12	49	524
TREC fbis	3.47	788	3911	4.26	98	740
TREC fr94	3.12	682	3196	3.49	84	689
TREC ft	3.13	761	3443	3.47	84	704
TREC latimes	2.99	742	3289	3.30	79	683

Table 5.4: PFOR-DELTA vs carryover-12

are stored in uncompressed form at the end of a disk block walking backwards. PFOR-DELTA is PFOR on the differences between subsequent column values.

PFOR and PFOR-DELTA can handle data distributions with outliers better than FOR, because they can represent outliers as exceptions, allowing b to stay low. This makes them better suited to compress *inverted lists*, which consist of increasing integer document identifiers that contain a term (or *gaps*, the differences between subsequent identifiers). Custom-built IR systems routinely employ compression of inverted lists [ZM06]. Recently in IR there is a trend towards compression schemes that sacrifice some compression ratio for better decompression speed [YDS09]. Carryover-12 is a recent example of such a compression scheme [AM05c].

Table 5.4 shows that on a number of IR datasets, PFOR-DELTA is 5-6 times faster than Carryover-12, at a cost of 15-20% lower compression ratio. Given the bandwidth provided by modern multi-disk hardware (i.e. 300-600MB/s), and IR compression ratios of around 3, a CPU needs to be able to produce 1-2GB/s of uncompressed data just to keep up with the disks. As the CPU needs to compute the IR ranking as well, decompression bandwidth must actually be significantly higher to prevent the IR system from being I/O bound. The main drivers behind the high speed in PFOR and PFOR-DELTA are their *vectorisable* algorithms, i.e. they decompress column values without control dependencies (if-then-else), using a technique called “patching”. By avoiding if-then-elses, modern CPUs are not slowed down by *branch-misprediction* events, and the vectorisation exposes data parallelism which modern super-scalar CPUs can exploit using *loop pipelining* and *speculative execution* to reach high IPC (Instructions Per Cycle) [ZHNB06].

5.5 Discussion

This chapter describes a number of optimisation techniques for the “database approach to sparse array computations” proposed in Chapter 4, with particular emphasises on the challenges and opportunities that array sparsity involves. The set of techniques presented can be further expanded to better reflect the difference between a prototype research system and more production-oriented software. New specialised operators can be introduced, for example to perform non-rectangular selections (diagonal, triangular, chessboard) that cannot be captured by the *RangeSel* operator. More sophisticated storage schemes can be used, for example to support data partitioning and replication, which in turn can provide efficient access patterns on multiple array dimensions (e.g. efficient access to both rows and columns of a matrix) and more opportunities for parallel processing. Optimisation techniques alone are not the whole story though. An important concept that this chapter aims to bring across is that the optimisation of inter-domain applications should be specialised to each stage of the transformation process from one domain to another, using the right technique at the right moment – exploit each domain's unique properties and propagate/translate information whenever possible. The effectiveness of this approach is the discussed in Chapter 6.

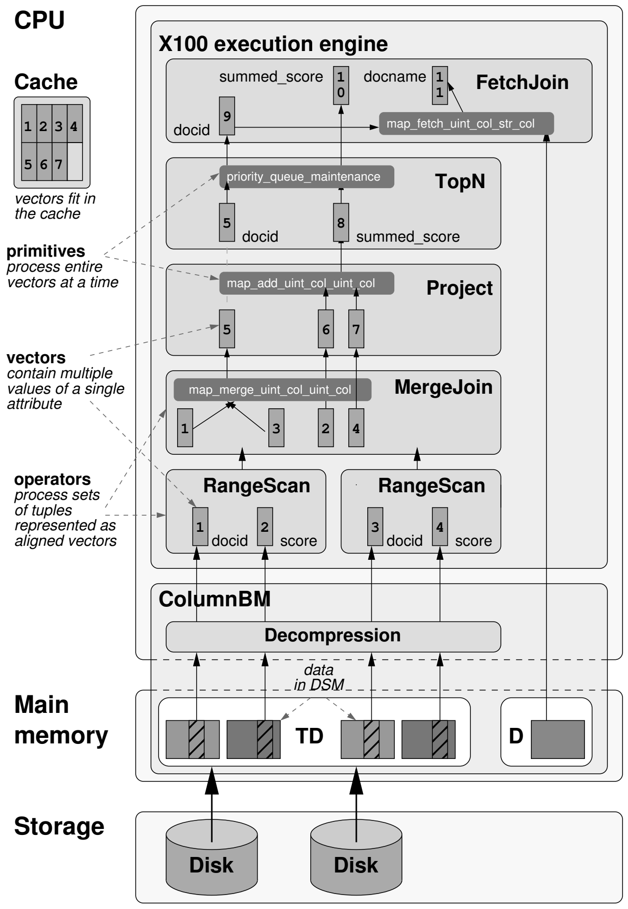

CPU
Cache
vectors fit in the cache
primitives process entire vectors at a time
vectors contain multiple values of a single attribute
operators process sets of tuples represented as aligned vectors
Main memory
Storage
X100 execution engine
FetchJoin
summed_score
docname
docid
map_fetch_uint_col_str_col
TopN
priority_queue_maintenance
docid
summed_score
Project
map_add_uint_col_uint_col
MergeJoin
map_merge_uint_col_uint_col
RangeScan
docid
score
RangeScan
docid
score
ColumnBM
Decompression
data in DSM
TD
D
Disk
Disk

Figure 5.5: MonetDB/X100 architecture (IR query example)

Extraordinary claims require extraordinary evidence.

- Carl Sagan, astronomer

6 Evaluation

To evaluate the work proposed in this thesis, we experiment on a number of scenarios with the proposed 'array-database' approach: *(i)* specify the given IR task in the array data-model (using the Matrix Framework for IR [RTK05]); *(ii)* express such specifications in SRAM syntax; *(iii)* obtain optimised database queries for such SRAM expressions; *(iv)* execute these queries on top of the MonetDB/X100 database engine [BZN05, ZBNH05]. Not all the steps above are analysed in detail in every scenario, as each of these targets a different evaluation aspect.

Aim of this chapter is to verify which conclusions can be drawn from such experiments in order to answer the questions posed in Chapter 1. In particular, the following quality indicators can provide insights on the success of this work.

Engineering simplification (Section 6.1) - *How does an interface based on the array data model simplify the database-powered implementation of parametrised search systems?* This question links back to Chapter 3, where the design principles of a Parametrised Search System (PSS) were introduced. Section 6.1 presents Spiegle, a real-life PSS implementation that instantiates ad-hoc database queries for given combinations of retrieval models (expressed declaratively as Matrix Framework for IR [RTK05] recipes using SRAM syntax) and data collections.

Flexibility (Section 6.2) - *How flexible is the proposed 'array-database' approach with respect to different IR tasks and/or data collections?* To show applicability of this approach to an IR task that goes beyond the classical keyword search, Section 6.2 discusses the implementation of a social network search model.

Efficiency (Section 6.3) - *Is the performance provided by this approach satisfactory? What is the importance of an optimised automatic translation compared to the raw speed that the underlining database engine could provide?* Section 6.3 presents the evaluation of a large scale text retrieval application based on a SRAM + MonetDB/X100 implementation of a retrieval model as defined in the Matrix Framework for IR [RTK05]. Efficiency, and in particular the impact that a flexible approach has on efficiency, are tested on a classical IR task, but challenging for its data size.

6.1 Spiegle: a PSS for XML retrieval

Chapter 3 introduces the concept of a *Parametrised Search System* (PSS). Distinctive feature of a PSS is to decouple search strategies from conceptual, logical and physical content representation, bringing to IR systems what the database field calls data independence and what De Vries has defined in [dV01] as content independence. This is achieved by carefully introducing abstraction layers for the declarative specification of search strategies, that stay as close as possible to the problem definition, while abstracting away the details of the physical organisation of data and content. In the same chapter, the array data-model is indicated as a well-suited data-access abstraction and the Matrix Framework for IR [RTK05] as a good framework for modelling IR in this abstraction. Chapters 4 and 5 detail on how to map the array data-model onto efficient query processing on top of a relational database engine.

Aim of this section is to show how this approach simplifies the engineering of a real PSS, Spiegle, which instantiates efficient database queries for specific combinations of search strategies and XML data collections.

6.1.1 PF/Tijah: a DB-powered XML retrieval system

PF/Tijah [HRvOF06] is a flexible XML search system implemented on top of a relational database system. It offers a good starting point towards a PSS, for two reasons: its relational backend natively implements data-independence and its clear separation between structure and content, from its application interface down to its application abstraction, facilitates the extension of the architecture layers responsible for content management.

PF/Tijah's layered system architecture, depicted in Figure 6.1a, uses the PathFinder (PF) XQuery compiler [AYBC$^+$06, BGvK$^+$06] to query by *structure*, as well as to construct the preferred result presentation. PathFinder translates the XQuery (non-ranked) part of a query into a relational query plan, independently from the ranking component of the system. The Tijah [BMR+06] XML retrieval system provides the *content* IR support, by processing the NEXI expressions in a query. NEXI [RT04] is a subset of XPath (it allows only descendant and self axis steps) that supports an additional `about()` clause, which ranks the selected XML elements by their content. The resulting query language provides a powerful way to customise (mainly) the structural aspects of a retrieval strategy.

PF/Tijah translates the NEXI expression into an internal logical query plan consisting of Score Region Algebra (SRA) operators [Mih06]. The SRA algebra operates on *regions*, portions of a collection with the following properties: start, end, name, type (e.g. note, attribute, word) and score. SRA includes operators that perform the following tasks on regions:

- *element and term selection* ($\sigma_{n=name,t=type}$);
- *element score computation* ($\sqsupset_p$), implementing the `about()` clause according to the desired retrieval model;
- *element score combination* ($\sqcap_p$, $\sqcup_p$), to compute `AND` and `OR` combinations;
- *element score propagation* ($\blacktriangleright$,$\blacktriangleleft$), to propagate scores to a common ancestor or descendant.

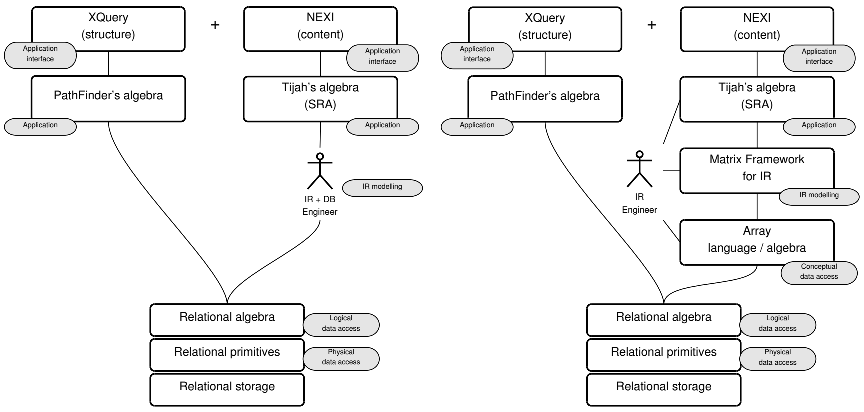
Consider the simplified excerpt of an XML database of scientific publications:

```
<PAPERS>
 <DOC>
  <text>
   <Abstract>IR as simple as cakes</Abstract>
   ...
  </text>
 </DOC>
 <DOC> ..... </DOC> ...
</PAPERS>
```

The following example retrieves *The title of documents written by John Smith, where the abstract is about "IR DB integration"*:

```
let $c := doc("papers.xml")//DOC[author = "John Smith"]
let $nexi := ".//text[about(.//Abstract, IR DB integration)];"
for $res in tijah-query($c, $nexi)
return $res/title/text()
```

The internal result of the XQuery expression `$c` is a non-scored region C. Then, the NEXI expression `$nexi` computes a scored result region R_{nexi} starting from region C. We show below the SRA formulation for region R_{nexi}, with



(a) PF/Tijah (b) Spiegle

Figure 6.1: Architectures of PF/Tijah and Spiegle

the following shorthands for the selection operator: $R^n_{\text{name}} = \sigma_{n=name,t=node}(C)$, $R^w_{\text{name}} = \sigma_{n=name,t=word}(C)$.

$$R_{\text{nexi}} = R^n_{\text{text}} \blacktriangleright ((R^n_{\text{abstract}} \sqsupset_p R^w_{\text{IR}}) \sqcap_p (R^n_{\text{abstract}} \sqsupset_p R^w_{\text{DB}}) \sqcap_p (R^n_{\text{abstract}} \sqsupset_p R^w_{\text{integration}}))$$

Each SRA operator implementation is given in terms of relational database queries (by a system engineer). SRA operators are parametric to some extent: several retrieval models are supported out-of-the-box for the ranking operator $\sqsupset_p$; scoring combination ($\sqcap_p$, $\sqcup_p$) can be performed using different functions, e.g. addition, multiplication, min, max; scoring propagation ($\blacktriangleright$,$\blacktriangleleft$) is similarly parametrised, with e.g. sum, average, weighted sum. Such parameters can be chosen for each query using an extra 'options' parameter to the `tijah-query()` function. However, the implementation corresponding to such parameters must be hard-coded in the system. The SRA layer is identified as the *application abstraction* layer in a PSS architecture. Although it provides a logical abstraction of the original user query, it does not contain the specification of the retrieval model to be used to accomplish such tasks.

6.1.2 PF/Tijah + Matrix Framework for IR + SRAM = Spiegle

While PF/Tijah provides a powerful query language to embed search functionality in data processing, it does not support easy customisation of the retrieval model.

Although advanced users may in principle modify the pre-defined mapping of SRA operators to relational query plans to implement new ranking functions, doing so is far from trivial. In other words, PF/Tijah supports the customisation of the structural aspects of various search strategies, but it is inflexible with respect to modifying the *content* aspects.

Spiegle overcomes this limitation in two steps. First, it supports the declarative specification of retrieval models, by employing the Matrix Framework for IR. This way, the search system engineer may implement information retrieval models at a high level of abstraction. Second, SRAM translates the specified retrieval model automatically into a relational query plan.

Figure 6.1a shows how in PF/Tijah the modelling and the implementation of retrieval models on top of the logical data-access abstraction are hard-coded as black-boxes in the system by a system developer, who is then required to be an IR *and* a DB engineer. Compare this with the desired Spiegle architecture in Figure 6.1b, where the modelling and the specification of retrieval models are open: the *IR engineer* employs the Matrix Framework for IR to define and modify dynamically retrieval models in a declarative array syntax, which are then translated to DB queries for the specific query-context automatically.

In the following, we detail how the Spiegle system uses its main components in practice. We first explain how the Matrix Framework can handle XML documents. Next, we see how to bootstrap, index and rank XML collections using the array data-model. The SRAM scripts shown are translated on-the-fly into database queries executed by the backend.

The Matrix Framework and XML data

The formalism of the Matrix Framework is not restricted to flat text search. Mapping the XML structure to the three dimensions location, document and term is all that is needed to apply the Matrix Framework to semi-structured collections. Each XML file can be seen as a collection of documents, where each distinct XML element represents a different one. The main difference from modelling plain text collections is that locations are now shared among different (nested) elements. Therefore, locations are defined as *term and XML tag* position in the file. Consider the following excerpt of the `papers.xml` collection of scientific publications, where location, document and term identifiers are annotated next to each piece of text for the reader's convenience, with `l`, `d` and `t` prefixes respectively:

```
<PAPERS>[l0,d0]
 <DOC>[l1,d1]
  <text>[l2,d2]
   <Abstract>[l3,d3] IR[l4,t0] as[l5,t1] simple[l6,t2] as[l7,t1] cakes[l8,t3]</Abstract>[l9]
   ...
  </text>[l40]
 </DOC>[l41]
 <DOC>[l42,d15] ..... </DOC>[l120] ...
</PAPERS>[l30000000]
```

Notice that: *(i)* the two `<DOC>` elements have different identifiers; *(ii)* the two `as` occurrences have the same term identifier `t1`; *(iii)* locations `l3` to `l9` belong to documents `d0`, `d1`, `d2` and `d3`.

Bootstrapping XML collections

A Matrix Framework representation of a collection is obtained from vectors L (location weight), D (document weight) and T (term weight), and matrices LD (location-document) and LT (location-term). For the example collection `papers.xml`, this corresponds to an SRAM script with the following definitions.

`papers.ram :`

```
// global information for collection "papers.xml"
nLocs=30000000 , nDocs=2000000 , nTerms=400000
L  = [ 1 | l < nLocs ]
D  = [ 1 | d < nDocs ]
T  = [ 1 | t < nTerms ]
LD = [nLocs,nDocs],  bool, sparse("0"), "LD_table"
LT = [nLocs,nTerms], bool, sparse("0"), "LT_table"
```

First, the length of each dimension is defined as `nLocs`, `nDocs`, `nTerms`. Then, vectors L, D and T are defined, using a constant weight 1. Finally, matrices LD and LT are declared as persistently stored in the database, as they are supposed to be the outcome of an earlier parsing and pre-processing phase on the XML file. For each persistent matrix, the dimensionality, the element type, the information on sparsity (LD and LT are both sparse with common value 0) and the name of the corresponding relational table are specified.

A script file similar to `papers.ram` is created automatically for each collection and it is included in every SRAM script that uses that particular collection. Notice that the script file above only contains SRAM *definitions* and no assignments (see Section 4.3), which result in simple in-memory declarations.

Array System Libraries

The uniform approach to IR tasks like indexing and retrieval that the Matrix Framework provides is easily operationalised by collecting the formulae

```
#include "MF_DocContext.ram"
bm25(d,k1,b) = sum( [ w(d,t,k1,b) * Q(t) | t ] )              // Okapi BM25
w(d,t,k1,b)  = log( nDocs / Tdf(t) ) * (k1+1) * DTf(d,t)
             / ( DTf(d,t) + k1 * (1 - b + b * Dnl(d) / avgDnl) )

langmod(d,a) = sum( [ lp(d,t,a) * Q(t) | t ] )                // Language Modelling
lp(d,t,a)    = log( a*DTf(d,t) + (1-a)*Tf(t) )
```

(a) `MF_RetrievalModels.ram`

```
// create index arrays as defined
// in the Matrix Framework for IR
#include "LinearAlgebra.ram"
DTnl := mxMult(mxTrnsp(LT), LD)
Dnl  := mvMult(mxTrnsp(LD), L)
Tnl  := mvMult(mxTrnsp(LT), L)
DT   := [ min(DTnl(d,t),1) | d,t ]
DTf  := [ DTnl(d,t)/Dnl(d) | d,t ]
Tf   := [ Tnl(t)/nLocs | t ]
```

(b) `MF_Indexing.ram`

```
// filter index arrays on the docs given in DX
DTnl = [ DTnl(d,t) * DX(d) | d,t ]
Dnl  = [ Dnl(d)    * DX(d) |   d ]
DT   = [ DT(d,t)   * DX(d) | d,t ]
DTf  = [ DTf(d,t)  * DX(d) | d,t ]
```

(c) `MF_DocContext.ram`

```
mxTrnsp(A)    = [ A(j,i) | i,j ]                          // matrix transposition
mxDiag(A)     = [ A(i,i) | i ]                            // matrix transposition
mxMult(A,B)   = [ sum([ A(i,k) * B(k,j) | k ]) | i,j ]    // matrix multiplication
mvMult(A,V)   = [ sum([ A(i,j) * V(j) | j ]) | i ]        // matrix-vector multiplication
mxNormRows(A) = [ A(i,j) / sum([ A(i,k) | k ]) | i,j ]    // normalise to 1 each row of a matrix
```

(d) `LinearAlgebra.ram`

Figure 6.2: SRAM libraries (excerpts)

presented in Section 3.4 in a system library composed by SRAM expressions (automatically made collection-dependent using a unique prefix for the collection's stored arrays). Figure 6.2 shows an excerpt of such a library, that is used in the subsequent paragraphs about indexing and retrieval of XML collections. One can observe that the SRAM expressions are an almost direct transcription of mathematical formulae to ASCII characters, which demonstrates the intuitiveness of array comprehensions as an IR query language.

Indexing XML collections

The indexing phase creates statistical information about a given collection. As described in Section 3.4, this entails computation of some of matrices that describe quantities and frequencies. Definition of such matrices in SRAM syntax is given in Figure 6.2d. Notice that this file contains array *assignments*, that create persistent arrays in the database. The SRAM script for indexing the example `papers.xml` collection becomes straightforward: load the collection's global definitions, followed by the generic index-creation assignments:

```
#include "papers.ram"       // global definitions for papers.xml
#include "MF_Indexing.ram"  // create index arrays
```

Ranking XML documents

Recall the example query of Section 6.1.1, where the NEXI expression `.//text[about(.//Abstract, IR DB integration)]` ranks the *"text" elements that contain an "Abstract" element about "IR DB integration"*. This structured IR part of the query is translated into SRA algebra. Selection of `text` and their containing `Abstract` elements is performed by *structure* operations implemented using PathFinder primitives. We call the resulting node-set the *ranking-context*, represented in the Matrix Framework by a binary vector DX, where 1 means that the corresponding document is in the ranking-context.

Ranking the `Abstract` elements of the *ranking-context* is performed by the *content* $\sqsupset_p$ SRA operation. This corresponds to an internal function with signature:

```
Function rank(collection, rankingContext, queryTerms, N,
              retrievalModel, param1, param2, ...) := rankedContext
```

This function turns the *ranking-context* of a given query into a top-`N` *ranked-context* against the specified query terms, by applying the desired retrieval model. Its body executes an SRAM script, customised at each call with the value of the current function parameters. The script of the example NEXI query above, with parameters `collection`="`papers`", `retrievalModel`="`langmod`" and `N`=20, corresponds to:

```
// global definitions for collection papers.xml
#include "papers.ram"
// ranking-context and query terms
DX = [nDocs],  bool, sparse("0"), "DX_table"
Q  = [nTerms], bool, sparse("0"), "Q_table"
// include retrieval model definitions
#include "MF_RetrievalModels.ram"
// retrieve top N documents using "langmod" (Language Modelling)
S = [ langmod(param1,param2,...) | d ]
D20 := topN( [S(d) | d<20], DESC )
S20 := S(D20)
```

First, global definitions for the collection `papers.xml` are loaded. Then, the *ranking-context* and the query terms are declared as persistent arrays, previously stored in the database by the system. Definitions of the available retrieval models are loaded (file `MF_RetrievalModels.ram`) and the selected one used to rank documents. Finally, the top 20 document identifiers and scores are computed and used together as the *ranked-context* returned by the SRA operator.

Figure 6.2a shows an excerpt of the available retrieval model expressions. These get customised to the current querying scenario by declaring the two arrays DX (*ranking-context*) and Q (*query-terms*) before each query is executed. The first line includes library file `MF_DocContext.ram` (Figure 6.2c). The array redefinitions in this script limit the document axes of index arrays to the current *ranking-context* by multiplying with the binary vector representing the *ranking-context* (the result of this multiplication corresponds to those portions of index arrays that contain information about the node-set to be ranked).

6.1.3 Discussion

The quality indicator addressed by the experiment discussed in this section is *engineering simplification.* More precisely, the question driving the Spiegle PSS example is: *does an interface based on the array data model allow the database-powered implementation of parametrised search systems, lowering the requirements in terms of database engineering expertise needed?.* The discussed PSS Spiegle builds upon PF/Tijah, which does provide a database implementation of the search strategies supported. However, the definition of such search strategies demands rather high database engineering skills. Spiegle replaces the hand-written database queries by automatic translations of search strategy specification, abstracting away physical details. In other words, the information retrieval engineer who writes search strategies for Spiegle no longer needs to cover the second role of (or be dependent on) a database engineer.

The task of the IR engineer working with Spiegle is additionally simplified by the usage of a formal framework like the Matrix Framework for IR. The unified approach that this provides to IR tasks facilitates the construction of search systems based on well-defined building blocks, preferably organised in a stack of array definition "libraries", from low-level linear algebra macros up to single-line instructions for indexing and retrieval. The Spiegle example also shows the flexibility of the Matrix Framework when it comes to handle transparently different kinds of collections, such as flat text or XML collections.

A possible direction for future work is indicated by the analysis of the LD (location-document) matrix defined in the Matrix Framework. The distribution of values 1 in this matrix is such that many consecutive locations belong to the same document (this is true for both flat text and XML collections). The relational representation of this matrix could employ a range-based encoding, rather than the *1 relational tuple = 1 array cell* encoding proposed in this thesis (see Section 4.5.1). For example, a series of tuples in the relation storing the LD matrix for a nested-documents collection (e.g. XML) could be encoded as two tuples only in the range-based representation:

locations 12-38 belong to both document 30 and 31

locations 39-54 belong to document 30

l	d	v
12	30	1
12	31	1
	⋮	
38	30	1
38	31	1
39	30	1
40	30	1
	⋮	
54	30	1

$\longrightarrow$

l_{start}	l_{end}	d_{start}	l_{end}	v
12	38	30	31	1
39	54	30	30	1

In the case of an LD matrix, the range-based representation would have a potentially large impact in terms of storage space needed and, more importantly, in terms of number of tuples to be processed, which can determine dramatic performance differences. The disadvantages that can be foreseen for this solution are however a higher complexity of the query plans and of the relational operators needed for array processing. For example, simple join operations would be replaced by expensive combinations of theta-joins with range-based join conditions (e.g. $A.d_{start} > B.d_{start}$), selections and projections. A similar encoding is used e.g. in the XQuery system [BGvK+06], which implements specialised relational operators for efficient processing. Because one of the design goals of SRAM is to rely on standard relational algebra as much as possible, we intentionally did not take the route of special operators implementation, with the only, well justified, exception of *IndexTable*. The choice of alternative relational encodings, with consequent alternative relational query mappings, is an optimisation problem that is considered a very promising direction for future work.

6.2 Personalised search in social networks

Aim of this section is to show the applicability of an array database approach to Information Retrieval tasks beyond the classical keyword search. The emphasis is put on the evaluation of how easily the problem can be expressed in terms of array computations, and on validating practical applicability of its automatic relational translation. For both aspects, achievements and hints for future improvements are highlighted.

6.2.1 Social networks

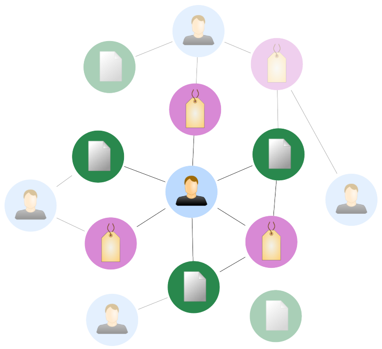
A social network is a map of the relationships between individuals. Social networking is the practice of active participation in the development of a social network. By being part of social networks and expanding their connections with others, individuals can boost their chances to find help for the tasks they aim to achieve (e.g. finding a job, making business, establish personal relationships). While such a concept has always existed in society (a family or a community are examples of social networks), the unparallelled potential of the Internet to promote such connections is only now being fully recognised and exploited, through Web-based groups established for that purpose.

Online social network services make virtual communities grow around the concepts of connection, affinity and similarity among users. However, the focus of each social network service can vary. Some focus on the interaction among users, the possibility of communicating and sharing personal information and are mostly used to keep in touch with or find new users considered "friends" (e.g. Facebook [Fac], MySpace [MyS]). Others rely on the concept of trust among connected users in some context (e.g. LinkedIn [Lin] helps to find and offer job opportunities by trusting the mutual professional opinion of connected users). Other social network services focus on specific topics and the knowledge of the virtual community created around them, in order to store, annotate, rate and suggest content of interest to their users (e.g. YouTube [You] for videos, LibraryThing [Lib] for books). These networks are referred to as *social content networks*.

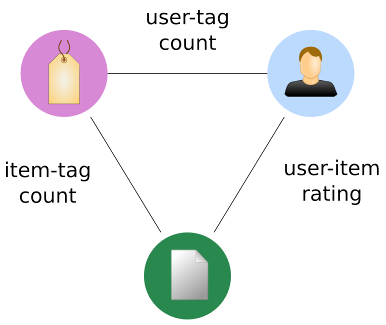
6.2.2 Search in social content networks

Social content network services focus on particular *items* (books, videos, images, etc). They provide a variety of mechanisms to help *users* find their way through the network and reach relevant content. A mechanism that has become popular in recent years is the possibility for users to annotate items with *tags*, which are short textual descriptions (often one or two words), and subsequently use these tags as query terms to find relevant items. For example, books can be assigned tags "fiction", "mystery" and "religion". By using such tags as a query, the result list would most probably include *The Da Vinci Code*, by Dan Brown, and other similar books.

Social networks whose main entities are users, items and tags can be visualised as a graph (Figure 6.3a). The act of tagging items creates connections between nodes. By assigning counts to connections, the following interpretations are possible (see Figure 6.3b):



(a) A User-Item-Tag graph.



(b) User-Item-Tag connections.

Figure 6.3: In the random walk model, the social content network is represented as a tripartite graph, containing users, items and tags as nodes. The edges between these entities are determined by ratings or tag counts.

User-Item indicates how many tags each user assigned to each item

User-Tag indicates how many items each user tagged with which tag

Item-Tag indicates how many users tagged each item with which tag

Intuitively, nodes in the graph whose connections have higher counts are more likely to be related. The same concept extends to multi-step paths in the graph: non-adjacent nodes are likely to be related when the sum of the counts along the path between the two is maximised. Therefore, browsing the network to find relevant entities corresponds to finding nodes that are reachable through high-score paths from a given node.

6.2.3 The random walk model

In this section, we implement a flexible social network search model based on Markov random walks, developed by Clements, De Vries and Reinders in [CdVR08]. Although the model captures a number of possible retrieval tasks, we focus on the task of personalised item recommendation: retrieve items that are relevant to a user's profile and to a specific query tag. We show how the model can be implemented on a database engine using SRAM. As in [CdVR08], we apply the model to a subset of LibraryThing data (see Section 6.2.4). We slightly simplify the model by omitting some weight parameters, to put more emphasis on its implementation.

A random walk on a graph is a special case of a Markov chain, a stochastic process with the Markov property: given the present state, the next state is given by a certain probability distribution and is independent of the past states. This probability distribution can be represented by a square *transition matrix* $A_{i,j}[nN, nN]$, with $\forall i : \sum_j A(i,j) = 1$, where nN is the number of nodes in the graph. Each element $A(i,j)$ contains the probability of a one-step transition from node i to node j. Elements $A(i,i)$ in the diagonal of matrix A describe the probability of *self transitions*. Self transitions have the effect of slowing down the diffusion of the walk through the network. While transition matrix A describes transition probabilities during the random walk, *state vector* $V_i^{(n)}[nN]$, with $\sum_i V^{(n)}(i) = 1$, describes the probability distribution of being at a certain node at step n. Transition and state probability distributions are the two ingredients used in the implementation of a random walk model.

When representing a social content network as a transition matrix, nodes referring to the same entity can be grouped together for convenience (changing the order in which nodes are represented does not influence the connections among them). Values for both axes i and j of matrix $A_{i,j}[nN, nN]$, as well as values for axis i of vector $V_i^{(n)}[nN]$, refer to IDs of users, items and tags, in this order:

$$\underbrace{[\underbrace{0, \ldots, (nU-1)}_{nU}, \underbrace{nU, \ldots, (nU+nI-1)}_{nI}, \underbrace{(nU+nI), \ldots, (nU+nI+nT-1)}_{nT}]}_{nN},$$

where nU, nI and nT are the number of users, items and tags in the network, respectively, and $nU + nI + nT = nN$. Figure 6.7 visualises the random walk for a personalised search (see Section 6.2.6), which finds items that are relevant to both a user and a tag.

Query entities can be selected by specifying with known probabilities the initial state of the graph, i.e. by assigning, in the initial state vector $V^{(1)}$, high probabilities to query nodes and 0 to all the remaining nodes. At any step n in the random walk, given a state vector $V^{(n)}$ and a transition matrix A, the multiplication $V^{(n+1)} = V^{(n)}A$ computes the new state vector at step $n+1$. This iterative process stops after a number *nSteps* of steps. The lower this number of steps, the more the final ranking is influenced by the probabilities in the initial state vector $V^{(1)}$ rather than by the background distribution of the network. Section 6.2.7 discusses how to estimate an optimal value for *nSteps*. Multi-step probabilities can be found by repeating the multiplication $V^{(n+1)} = V^{(n)}A$ or multiplying the initial state vector with the n-step transition matrix A^n (matrix power): $V^{(n+1)} = V^{(1)}A^n$. Although this approach requires only one vector-matrix multiplication at query time for an n-steps random walk, it has two disadvantages:

first, the density of matrix A^n, δ_{A^n}, grows quickly to 1, even for small values of n (we observed this already when $n = 5$), which makes matrix A^n not manageable for non-trivial vales of nN; second, this solution is less flexible, as it would require a matrix A^n to be stored for every random walk's length available at query time.

6.2.4 LibraryThing

LibraryThing is a prominent social content network service for storing and sharing personal library catalogues. Users of LibraryThing can store books and provide public and private annotations. In particular, they can rate each book and assign it tags. The resulting *user*, *item*, *tag* entities form a tripartite graph that is suitable to be browsed using the random walk model discussed in Section 6.2.3.

We use annotation data relative to the same subset of the entire LibraryThing network as in [CdVR08] (crawled in July 2007). The authors of [CdVR08] have collected a trace from the LibraryThing network, containing 25,295 actively tagging users. After pruning this data set they retained 7279 users that have all supplied both ratings and tags to at least 20 books. They removed books and tags that occur in less than 5 user proles, resulting in 37,232 unique works and 10,559 unique tags. This pruned data set contains 2,056,487 UIT relations, resulting in a density of $7.2 \cdot 10^7$. The derived UI, UT and IT matrices have size and density as shown in Table 6.1b and all show the power-law behaviour common to social networks [HRS07]. After a training phase as described in Section 6.2.7, an optimal random walk length for personalised search on this network is estimated to be 13 (see Table 6.1a).

Specialisation of the random walk model for the LibraryThing network here considered is obtained by simply including the definitions in Figure 6.4 before the execution of scripts in figures 6.6 (indexing, see Section 6.2.5), 6.8 (personalised search, see Section 6.2.6) and 6.10 (parameter estimation, see Section 6.2.7).

6.2.5 Indexing: building the transition matrix

Figure 6.5a shows how matrix A can be composed by concatenating smaller matrices that represent specific portions of the graph. For example, all connections between user and item nodes are captured by matrix UI. Matrices UI, UT and IT come from user annotations:

User-Item (UI) matrix contains user-item ratings

User-Tag (UT) matrix contains user-tag counts

Item-Tag (IT) matrix contains item-tag counts

Table 6.1: Parameters and arrays used in a random walk over a subset of the LibraryThing social network, as described in Section 6.2.4.

(a) Parameters.

Symbol	Description	Value
nU	# users	7279
nI	# items	37232
nT	# tags	10559
nN	# nodes	55070
$nSteps$	# steps	13
θ	item/tag weight	0.6

(b) Arrays.

Array	$\mathcal{S}$	ε	δ
UI	$[nU, nI]$	0	$2.8 \cdot 10^{-3}$
UT	$[nU, nT]$	0	$5.2 \cdot 10^{-3}$
TI	$[nT, nI]$	0	$2.0 \cdot 10^{-3}$
A	$[nN, nN]$	0	$1.3 \cdot 10^{-3}$
$V^{(1)}$	$[nN]$	0	$3.6 \cdot 10^{-5}$

```
// Dimensions of LibraryThing annotations (number of users, items,tags)
nU = 7279
nI = 37232
nT = 10559

// Dimension of state vector and transition matrix
nN = nU + nI + nT

// Collaborative tagging output: relations between Users, Items, Tags
UI = ([nU,nI], int) sparse(0, 2.8E-3) "UI_LibraryThing"
UT = ([nU,nT], int) sparse(0, 5.2E-3) "UT_LibraryThing"
IT = ([nI,nT], int) sparse(0, 2.0E-3) "IT_LibraryThing"

// Optimal parameters found for personalised search on LibraryThing data
nSteps = 13
theta = 0.6
```

Figure 6.4: `LibraryThing_Defs.ram`

Note that User-Item "preference" relations were described by the number of tags assigned to items from users (see Section 6.2.2). Explicit user ratings per item are however available in our dataset and are expected to give better indications of user preferences than tag counts.

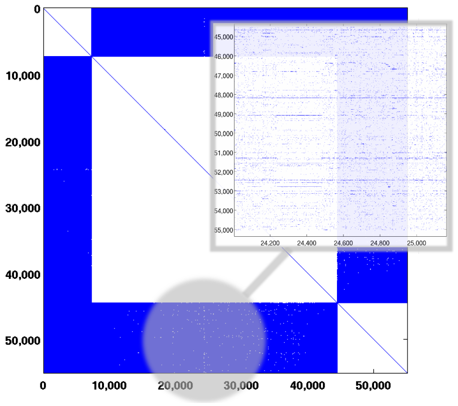
By transposing matrices UI, UT and IT, matrices IU, TU and TI are obtained. Matrices UU, II and TT are identity matrices (diagonal matrices of 1's) representing self-transition edges in the graph. Figure 6.6 shows the SRAM script that performs such operations. Each row of matrix A is finally normalised to sum to 1, so that it contains transition probabilities.

Figure 6.5b shows the distribution of non-ε elements in matrix A relative to the subset of LibraryThing data introduced in Section 6.2.4. Such a clustered distribution could be better exploited by using more sophisticated storage and processing schemes. For example, one could see matrix A as the combination of 4 sparse matrices (1 per side) with relatively high density and 3 extremely

$$\begin{array}{c|ccc}
 & \mathsf{U} & \mathsf{I} & \mathsf{T} \\
\hline
\mathsf{U} & \boxed{UU} & \boxed{UI} & \boxed{UT} \\
\mathsf{I} & \boxed{IU} & \boxed{II} & \boxed{IT} \\
\mathsf{T} & \boxed{TU} & \boxed{TI} & \boxed{TT}
\end{array} = A$$

$$\forall\ \text{row}: \sum_{\text{columns}} = 1$$

(a) Transition matrix A for a random walk on a social content network consists of matrices UI, UT and IT.



(b) Visualisation of non-ε elements of matrix A for LibraryThing data.

Figure 6.5: Construction of matrix A and its visualisation for LibraryThing data.

sparse matrices (the ones on the diagonal), and apply separately different mapping strategies. Other properties of array data, like symmetry, could be effectively exploited. Notice however, that matrix A is not symmetric due to the row-wise normalisation. Further investigations in these directions are part of future work.

6.2.6 Personalised search

Depending on the combination of the entities selected as starting nodes in a random walk model, different tasks are possible in a social content network service (e.g. "find more items like this", "find alike users", "find experts on a certain topic". See [WCY$^+$10] for a complete taxonomy). Personalised search is the task of finding items relevant to both a given tag and a user profile: "find items related to this tag, taking my personal preferences into account".

Figure 6.8 shows the SRAM script for performing a personalised search. The script is completely network-independent, and specialised to be used on the LibraryThing network by the inclusion of specific definitions, at Line 2.

Model parameter $\theta \in [0, 1]$ is used to tune the influence of the query user profile versus the query tag. Recall that the probabilities in the initial state vector must sum to 1. At Line 7, query user and tag are assigned values $1 - \theta$ and θ, respectively, in the initial state vector. When θ is set to 0, the query tag will be ignored and the most relevant items to the query user will be retrieved.

```
#include "LinearAlgebra.ram"
#include "LibraryThing_Defs.ram"

// UI, UT, IT come from network-specific definitions
// Declare their transpositions
IU = mxTrnsp(UI)
TU = mxTrnsp(UT)
TI = mxTrnsp(IT)

// Self-transition diagonal matrices
UU = [ int(x==y) | x<nU, y<nU ]
II = [ int(x==y) | x<nI, y<nI ]
TT = [ int(x==y) | x<nT, y<nT ]

// Build row-blocks of transition matrix A
Au = mxTrnsp(UU ++ IU ++ TU)
Ai = mxTrnsp(UI ++ II ++ TI)
Au = mxTrnsp(UT ++ IT ++ TT)

//  Concatenate the row-blocks, normalise rows, store matrix A
A := mxNormRows(Au ++ Ai ++ At)
```

Figure 6.6: `randomWalk_Index.ram`

When θ is set to 1, the profile of the query user will be ignored, yielding a standard user-independent, tag-based, popularity search.

The random walk is implemented by repeatedly updating the state vector through multiplication with transition matrix A (Line 12). The length of the walk is determined by *nSteps*. At the end of the loop that implements the walk, item relevance probabilities are to be found in the corresponding portion of vector $V^{(nSteps)}$. Figure 6.7 visualises this process.

At Line 16, only those items from vector $V^{(nSteps)}$ that have been assigned the selected query tag are retained. Items that have not been assigned such a tag are removed only at the end of the random walk because even though they do not qualify the tag selection, their connections in the network can provide useful information to reach relevant content.

Analysis of the SRAM implementation

Figure 6.9 shows the relational translation of the SRAM script in Figure 6.8 into the MonetDB/X100 query language. The following sections highlight some of the key optimisation strategies used, as well as some of the aspects that need to be further investigated.

Creation of very sparse arrays. The SRAM expression at Line 1 of Figure 6.9, that initialises the state vector $V^{(1)}$ with query user and tag IDs, creates

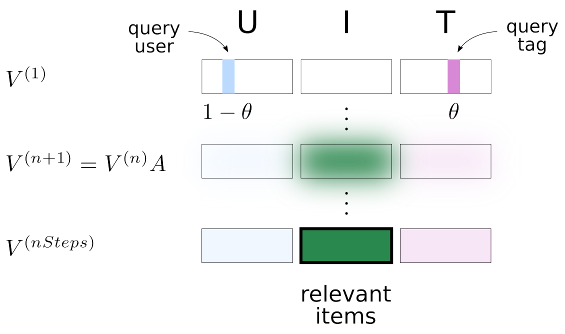


Figure 6.7: The personalised search task.

```
#include "LinearAlgebra.ram"
#include "LibraryThing_Defs.ram"

// Select the query user and tag in the initial state vector
// Query ID's 'userID' and 'tagID' come from an external application
V := [ if       (x == userID)         then (1-theta)
       else if (x == (nU+nI+tagID)) then theta     else 0 | x<nN ]

// Random Walk
repeat(nSteps)
  // Take one step
  V := vmMult(V,A)
end repeat

// Retain only items from V that have been assigned the tag 'tagID'
Items := [ if (IT(i,tagID) != 0) then V(nU+i) else 0 | i<nI ]
```

Figure 6.8: `LibraryThing_Search.ram`

a very sparse array (only 2 in nN non-ε values). This is mapped onto relational algebra by selecting the two corresponding values of attribute `x` from those created by the *IndexTable*($[nN]$) operator (implemented with `Array` in MonetDB/X100). While such a solution does not pose severe efficiency issues with relatively small arrays such as those for the LibraryThing subset used here, it could become less viable when large *IndexTable*($\mathcal{S}$) relations need to be created. A cheaper alternative, when a sparse array only has a few known non-ε values, would be to insert those values directly into an empty table.

Avoiding materialisation in sparse arrays triggers relational simplifications. One can notice how the initialisation of vector $V^{(1)}$ has lost one of its original `if-then-else` constructs during the translation (compare definitions starting at lines 1 and 3). The reason is that the last `else` branch of the SRAM

```
 1 // V := [ if      (x == userID)        then (1-theta)
 2 //       else if (x == (nU+nI+tagID)) then theta     else 0 | x<nN ]
 3 V :=  Project(
 4         Select(
 5           Array(x = dim(nN)),
 6           (x == userID) || (x == (nU+nI+tagID))
 7         ),
 8         [x, v = ifthenelse(x == userID, 1.0-theta, theta)]
 9       )
10
11 repeat(nSteps)
12   // V := vmMult(V,A)
13   V :=  Aggr(
14           Project(
15             MergeJoin(V, A, V.x == A.x ),
16             [V.x, A.y, v = V.v * A.v]
17           ),
18           [x=A.y], v = sum(v)
19         )
20
21 // Items := [ if (IT(i,tagID) != 0) then V(nU+i) else 0 | i<nI ]
22 Items :=  Project(
23             MergeJoin(
24               Select( IT, IT.t == tagID ),
25               Project(
26                 Select( V,  (x >= nU) && (x < (nU + nI)) ),
27                 [i = V.x - nU, v],
28               )
29               IT.i == V.i
30             ),
31             [i = V.i, v = V.v]
32           )
```

Figure 6.9: `LibraryThing_Search.x100`

definition produces the value 0. Because $\varepsilon_V = 0$, the tuples corresponding to that branch need not be stored (and therefore computed), which turns the remaining `if-then` construct into a projection of the `then` branch over a selection on the `if` branch. The same simplification happens for the conditional expression at Line 21 of Figure 6.9, where the `else` branch needs no storage, which turns the `then` branch into a projection and the `if` branch into a selection (a selection over a cartesian product in this case, i.e. a join).

Not needing to store, and therefore to compute, the results of the `else` branch propagates its benefits to the sub-expressions upon which this depends. At Line 21 of Figure 6.9, the values that would make true the `else` branch are identified by the boolean expression `(IT(i,tagID) == 0)`. Because those values correspond to ε elements, they are not stored in the `IT` table. In case the `else` branch had needed storage, they would have had to be created by an expensive $\mathsf{IndexTable}(\mathcal{S})$ operation.

Arithmetic optimisation. A number of simplifications that are crucial for efficient processing of sparse arrays are enabled by the arithmetic optimisation that takes place during the array to relational algebra mapping (see Section 5.2.2).

The boolean condition in the `if` branch of the conditional expression at Line 21 of Figure 6.9, `(IT(i,tagID) != 0)`, is true for every tuple stored in table `IT`, as $\varepsilon_{IT} = 0$. This allows to simplify the translation of the original condition from `Select( IT, (IT.t == tagID) && (IT.v != 0) )` to `Select( IT, IT.t == tagID )`.

A single step in the random walk (Line 12 of Figure 6.9) is performed by a vector-matrix multiplication, which is expressed as an SRAM macro (see Figure 6.2d). This is a setting similar to Example 5.14 on page 120, where we apply the principle of arithmetic optimisation (see Section 5.2.2). All the ε elements from both input arrays are recognised not to influence the final result at any stage in the query tree, which makes it possible to evaluate the expression based on the stored tuples only. This arithmetic optimisation together with a careful relational mapping optimised for sparse arrays (see Section 5.1.2), allow for a final translation into MonetDB/X100 syntax that is completely free from any overhead related to sparse array processing.

Choice of physical operators. Physical storage layout has a high impact on the choice of relational algorithms and consequently on the overall efficiency. The (repeated) vector-matrix multiplication at Line 12 of Figure 6.8 illustrates the problem. In the first loop iteration, the join condition (`V.x == A.x`), at Line 15 of Figure 6.9, is defined on the attributes on which the two tables are sorted (the major axis for both arrays), which triggers the usage of a fast `MergeJoin` algorithm. The `MergeJoin` operator processes tuples sequentially and preserves their order in output (here on `V.x` or, equivalently, `A.x`).

Unfortunately, this property cannot be further exploited, as the aggregation that follows the join happens on a different attribute (`A.y`). This forces the usage of a hash-based aggregation operator, which can be much more expensive than one optimised for exploiting tuple order or grouping. Moreover, a hash-based aggregation outputs tuples in no defined order, which limits the optimisation opportunities of following operators.

This is indeed the case when vector $V^{(n)}$ is used to compute the new vector $V^{(n+1)} = V^{(n)}A$ at the next iteration in the loop. In order to exploit the `MergeJoin` algorithm, table `V` needs to be sorted on its axis `x` before being used for the next iteration. This happens implicitly here, as the example stores array V at every iteration, which implies sorting table `V` on its major axis (the default policy of SRAM before every persistent storage). An alternative plan would use a much slower `HashJoin` algorithm for the join, but save the cost of sorting table `V` before using it for the next iteration. By not forcing table `V` to be sorted, such a plan would be considered by an optimiser.

6.2.7 Parameter estimation

Every combination of retrieval task and social content network gives the best results with a different instantiation of the parameters that influence the random walk model. We focus in this section on the estimation of parameters for personalised search (see Section 6.2.6) through a parameter sweep on a subset of the LibraryThing network (see Section 6.2.4). The parameters to be estimated are:

- n, that determines the optimal length of the random walk
- θ, that balances the influence of query user and query tag

The estimation process can be briefly described as follows: for every combination of parameters to be estimated, perform a personalised search of items belonging to a validation set and evaluate the performance of that search; the parameter combination that yields the best performance on the validation set is assumed to be optimal for that network.

More in detail, the process consists of three main steps:

1. The validation set is defined by holding out part of the users, with their annotations, from the network. These are called the test users, while the tags and the items relative to each of them are called the test tags and the test items, respectively.
2. For each test user tu, for each test tag tt used by tu, for each combination of the parameters to be estimated, a random walk on the entire network is performed, with tu and tt as initial nodes in the query.
3. The retrieved items (for each test user, test tag and parameter combination) include the test items of the validation set. By retaining those only and comparing their ranking to the the ground truth in the explicit UI ratings, it is possible to evaluate the performance of each random walk. The optimal parameters for n and θ are found by averaging the performance over all test tags and users.

Figure 6.10 shows the SRAM script that focuses on the second step only. The script is completely network-independent, specialised to be used on the LibraryThing network only by the inclusion of specific definitions, at Line 2. The extraction of a validation set from the entire network (step 1) is omitted. In the script, two macros, `isTestUI(u,i)` and `isTestUT(u,t)`, are used to determine whether given user-tag or user-item combinations belong to the validation set, independently of how this is chosen. These macros are assumed to be defined externally.

```
 1 #include "LinearAlgebra.ram"
 2 #include "LibraryThing_Defs.ram"
 3
 4 // Test values for theta
 5 nth    = 10
 6 Thetas = [ x / nth | x<nth ]
 7
 8 // Select a test user and tag per initial state vector
 9 V := [ [ if      isTestUT(u,t)
10          then    if      (x == u)             then (1-Thetas(th))
11                  else if (x == (nU + nI + t)) then Thetas(th)     else 0
12          else    0
13        | x<nN ]
14      | u, t, th ]
15
16 // Random Walk
17 Items := [ [ 0 | i<nI ] | n<maxSteps, u<nU, t<nT, th<nth ]
18 repeat(step,maxSteps)
19   // Take one step per state vector
20   V := [ vmMult(V(u,t,th),A) | u, t, th ]
21
22   // Select Test user-item-tag combinations
23   Items := [ if (n == step) then [ if (isTestUT(u,t) && isTestUI(u,i))
24                                    then V(u,t,th)(nU+i) else 0
25                                  | i<nI ]
26              else Items(n,u,t,th)
27            | n, u, t, th ]
28 end repeat
```

Figure 6.10: `LibraryThing_ParEst.ram`

Analysis of the SRAM implementation.

The following sections discuss some of the highlights and issues of both the SRAM script for parameter estimation (Figure 6.10) and its relational translation into the MonetDB/X100 query language (Figure 6.11).

Multiple simultaneous personalised searches. The SRAM script that performs step 2 is very similar to the one for personalised search (see Figure 6.8). However, several random walks take place instead of only one, which makes the arrays involved have higher dimensionalities. At Line 9 of Figure 6.10, one state vector per test user, test tag associated to that user, and theta value (here ranging from 0 to 1 in steps of 0.1) combination is initialised. For each initial state

```
// Figure~6.10, Line~9
V :=  Project(
        MergeJoin(
          FetchJoin(
            Select(
              Array(u = dim(nU), t = dim(nT), th = dim(nth), x = dim(nN)),
              (x == u) || (x == (nU + nI + t))
            ),
            Project( Array(ith = dim(nth)), [ theta = ith / nth ] ),
            th
          ),
          Select(UT, isTestUT(UT.u,UT.t)),
          (u == UT.u) && (t == UT.t)
        ),
        [ u, t, th, x, v = ifthenelse(x == u, (1-theta), theta ]
      )

repeat(step,maxSteps)
  // V := [ vmMult(V(u,t,th),A) | u, t, th ]
  V :=  Aggr(
          Project(
            HashJoin(V, A, V.x == A.x ),
            [V.u, V.t, V.th, V.x, A.y, v = V.v * A.v]
          ),
          [u = V.u, t = V.t, th = V.th, x=A.y], v = sum(v)
        )
  // Figure~6.10, Line~23
  Items :=
     Union(
      Project(      // then output
        HashJoin(   // then branch
          MergeJoin(
            Select(UI, isTestUI(UI.u,UI.i)),
            Select(UT, isTestUT(UT.u,UT.t)),
            UI.u == UT.u
          ),
          Project(
            Select(V, (x >= nU) && (n < (nU + Ni))),
            [u, t, th, i = x - nU, v]
          ),
          (UT.u == V.u) && (UT.t == V.t) && (UI.i == V.i)
        ),
        [ n = step , V.u , V.t, V.th, V.i, V.v ]
      )
      Project(      // else output
        MergeJoin(
          Select(Array(n = dim(maxSteps)), n != step)  // if (n != step)
          Items,    // else branch
        ),
        [ Items.n, Items.u, Items.t, Items.th, Items.i, Items.v ]
      )
    )
```

Figure 6.11: `LibraryThing_ParEst.x100`

vector, a random walk long $maxSteps$ steps is executed and the item ranking at every step n saved to array $Items_{n,u,t,th}[maxSteps, nU, nT, nth]$.

Nested arrays for increased expressiveness. Line 20 of Figure 6.10, which performs a single step (for all combinations) in the random walk, shows one of the advantages of allowing nested arrays in the syntax (see Section 4.3.6). The type of array $V_{u,t,th}$ is $([nU, nT, nth], ([nN], dbl))$. For every combination of axes

u, t and th, the expression `V(u,t,th)` returns a vector (a 1-dimensional array), that is fed to macro call `vmMult(V(u,t,th),A)` for a vector-matrix multiplication. Vector V can also be flattened into a 4-dimensional array $V_{u,t,th,x}$, with type $([nU, nT, nth, nN], dbl)$. The underlying relational representations for the nested and flattened versions of V are identical. However, the flattened version would reduce significantly the benefits of having a standard library of linear algebra macros. In our example, in order to use macro `vmMult()`, which expects a 1-dimensional array as a first parameter, a much less straightforward selection on V would be needed before the macro call can take place: `vmMult([ V(u,t,th,x) | x ],A)`.

Access patterns. The expression at Line 20 of Figure 6.10 raises however the issue of how to automatically disentangle a declarative array syntax and the choice of efficient access patterns to array data. Nested array $V_{u,t,th}$, with $\mathcal{T}_V = ([nU, nT, nth], ([nN], dbl))$, is physically stored as a flat array $V_{u,t,th,x}$, and therefore the associated table is sorted on axes `u`,`t`,`th` and `x` in this order. SRAM expression `vmMult(V(u,t,th),A)` returns an array whose relational representation is a non-ordered subset of tuples in table `V`. For this reason, the relational translation of the vector-matrix multiplication cannot use a `MergeJoin` algorithm, but has to use a `HashJoin` algorithm instead (see Line 22 of Figure 6.11). One could imagine reformulating the script in such a way that V is a vector of 3-dimensional arrays ($\mathcal{T}_V = ([nN], ([nU, nT, nth], dbl))$), rather than a 3-dimensional array of vectors ($\mathcal{T}_V = ([nU, nT, nth], ([nN], dbl))$, to have axis x as the major one. However, this would prevent the usage of macro call `vmMult(V(x),A)`, as the expression `V(x)` would return a 3-dimensional array. All the computations that are based on array V would have to be transformed. For example, instead of computing one vector-matrix multiplication for each combination of axes u,t and th, one would have to compute three-dimensional intermediate steps of one vector-matrix multiplication. Compare the two resulting options for Line 20 of Figure 6.10:

```
// When T_V = ([nU,nT,nth],([nN], dbl))
vmMult(V,A)   = [ sum([ V(x) * A(x,y) | x ] ) | y ]
V = [ vmMult(V,A) | u,t,th ] =
  = [ [ sum([ V(u,t,th)(x) * A(x,y) | x ] ) | y ] | u,t,th ]

// When T_V = ([nN],([nU,nT,nth], dbl))
v3mMult(V3,A) = [ [ sum([ V3(x)(a,b,c) * A(x,y) | x ] ) | c,b,c ] | y ]
V = v3mMult(V,A)
  = [ [ sum([ V(x)(u,t,th) * A(x,y) | x ] ) | u,t,th ] | y ]
```

The disadvantage of the second solution is that macro `v3mMult()` is far less generic than macro `vmMult()`. While macro `vmMult()` corresponds to a well-defined algebraic pattern (the multiplication of a vector and a matrix) and can be used inside comprehension expressions of any dimensionality, macro `v3mMult()` can only be used with a vector of 3-dimensional matrices.

Although array computations can be expressed declaratively in SRAM syntax, this is not completely disentangled yet from some of the physical details that can influence the efficiency of relational translations. This example shows that the optimal order of tuples of array tables still needs, in some cases, to be steered manually at syntax level.

The NDCG evaluation in SRAM. The final evaluation of the retrieved item ranking (step 3) is omitted from Figure 6.10. In [CdVR08], the Normalised Discounted Cumulative Gain (NDCG) [JK02] is used to perform such an evaluation. The computation of NDCG includes a cumulative sum, which is expressible in SRAM as

```
cs(V) = [ sum([ V(i) | i<j+1 ]) | j<len(N,0) ]
CSI  := cs(Items(n,u,t,th))
```

and translated to the following MonetDB/X100 query:

```
CSI :=  Aggr(
          Join(
            V,
            Array(j = dim(nI)),
            V.i <= j
          )
          [ V.n, V.u, V.t, V.th, j ],
          sum(V.v)
        )
```

However, the performance of this solution is dramatically worse than what a native support in the database engine for this type of functions would provide.

The number of relevant items found varies for each combination of n, u, t and th. In order to be able to compare the Discounted Cumulative Gain of different item rankings, these have to be normalised. The normalisation phase includes sorting the sub-vector of relevant items. SRAM syntax allows this operation on vectors only (see Section 4.3.6), which is not a problem when nested arrays are used:

```
I := [ sort(Items(n,u,t,th), DESC) | n, u, t, th ]
```

However, recall that nested arrays are flattened before being translated into relational algebra. Therefore, sorting each vector contained in the 4-dimensional array *Items* corresponds to sorting separately each group of table `Items` identified by attributes `n`, `u`, `t` and `th`. Such a per-group sort is not part of the standard set of relational operators available in database engines. The problem is identified in [CCG00], where per-group top-N processing is explicitly supported in a standard RDBMS with extensions at all levels (SQL interface, relational internal representation, algorithmic implementation).

6.2.8 Discussion

The scenario presented in this section aims to show that the proposed framework can support other IR tasks than classical keyword search. Personalised search in social networks is implemented using a random walk approach, which makes extensive use of the array data model. It is worthwhile summarising a few elements of interest that arise from this exercise.

While it may be hard to match the final MonetDB/X100 queries shown in this section with the original random walk tasks, the SRAM definitions that describe the raw data and the indexing and search processes are remarkably concise and close to the elegance of their respective mathematical formulation. Even the definition that describes parameter estimation, which is inherently more complex due to the higher-dimensional nature of the problem (parameter estimation can be seen as a multi-random walk process), remains compact and fairly easy to read. This is mainly due to the abstraction of nested arrays, which allows for simplified programming at no additional computational cost.

This experiment clearly indicates that some of the limitations already identified in Chapter 4 translate indeed into reduced expressive power in practice, which can often be circumvented at the cost of efficiency and / or elegance. Most notably, the lack of a an explicit looping mechanism in the relational algebra makes non-trivial iterative tasks hard to express efficiently (as a work-around, looping may be handled externally by a scripting language). Per-group sorting (used in the Normalised Discounted Cumulative Gain computation) is another operation not directly supported in the relational algebra. A possible, though inefficient, work-around selects groups into new arrays, which are then sorted individually and finally recombined. Finally, the current implementation takes a simplistic approach for what concerns access pattern efficiency, in that SRAM syntax in not completely independent of physical storage patterns and their inherent efficiency (ordered access is guaranteed on the first dimension of each array). This topic, not limited to array databases, needs further attention, for example by allowing redundant storage to accommodate multiple access patterns, similar to how relational optimisers use index structures for the same purpose. A dedicated optimisation phase could try to find the minimal set of physical representations to choose from at each computation, so that the number of expensive access patterns is minimised. This problem is not specific to the array nor to the relational domains. Tools like Matlab [Mat], although very optimised for fast array computations, exhibit extremely different performance when, say, the same matrix is accessed by row or by column (matrices in Matlab are stored column-wise, which makes accesses by row perform expensive hops in memory). Similarly, some information retrieval tools that use inverted files rely on multiple copies of their indices, differently ordered to optimise access speed to different entities, e.g. get documents given a term, or get terms given a document.

6.3 Large scale text retrieval

This section illustrates how we used SRAM, following the indications of the Matrix Framework for IR [RTK05], to build a text IR application on top of the MonetDB/X100 database engine [BZN05, ZBNH05]. Our main purpose with this application is to demonstrate the efficiency with which IR retrieval models can be specified and implemented using the array data model interface on an adequately fast database engine. To validate our results, we tested our application on a standardised large scale efficiency task, the TREC TeraByte track (TREC-TB) [CSS05]. Section 6.3.1 describes the setup of this IR application, including the IR methods we selected for it. Thereafter, Section 6.3.2 discusses experiments and achieved results.

6.3.1 TREC-TB

TREC TeraByte track [CSS05] has introduced a task to evaluate IR system efficiency on ranking a (back then) large web-crawled collection of documents (the GOV2 collection). This data set consists of 25 million web documents, with a total size of 426GB. System efficiency is measured by total execution time of 50,000 queries. Effectiveness is evaluated by early precision (*p@20*) on a subset of 50 pre-selected queries for which relevance judgements are available.

Okapi BM25 as an array query

We selected the Okapi BM25 [RWB98] formula as the IR retrieval model for these experiments. Note that we select BM25 just for its ubiquity and aptness for TREC-TB.

The BM25 document scoring formula is:

$$\omega_{D,T} = \log\left(\frac{n_D}{f_T}\right) \cdot \frac{(k_1+1)\cdot f_{T,D}}{f_{T,D} + k_1 \cdot ((1-b) + b \cdot \frac{|D|}{avgdl})} \tag{6.1}$$

$$S^{(D)}_{BM25} = \sum_{T \in Q} \omega_{D,T} \tag{6.2}$$

The constants used in this formula are shown in Table 6.2 and all arrays used in Table 6.3. Note that these arrays are defined as in the Matrix Framework for IR (see 6.3.1 for more details), although the naming conventions can differ slightly from [RTK05]. The SRAM script that computes the top 20 results according to the BM25 document score model is:

```
1 w(d,t) = log( nDocs / F(t) ) * ((k1 + 1) * TD(t,d))
2                               / (TD(t,d) + k1 * ((1 - b) + (b * S(d) / avgdl)))
3 s(d)   = sum( [ w(d,Q(t)) | t ] )
4 D20 := topN( [ s(d) | d ], 20, DESC )
```

One can observe that the SRAM expression is an almost direct transcription of the mathematical formula to ASCII characters, which demonstrates once more the intuitiveness of array comprehensions as an IR query language. Array expressions at lines 1 and 3 correspond to (6.1) and (6.2), respectively. Arrays F_t, $TD_{t,d}$ and S_d contain document frequency for terms t, within-document term frequency for (t,d) pairs and document size (length) of documents d, respectively. Dense array Q^i_t enumerates query terms for each query Q^i. These query vectors are temporary objects, typically instantiated inside the query, using the explicit array constructor syntax defined in Section 4.3.6. Finally, the expression at Line 4 selects the 20 best scored documents and materialises their identifiers in array $D20$. Notice that the array of all scored documents is not stored persistently before selecting the top 20 documents. This allows sophisticated relational top-N implementations to perform an early-stop of the stream of tuples in input, limiting the cardinality of intermediate results in their sub-tree, under certain conditions [CK97].

TREC-TB data preparation

To parse the TREC-TB GOV2 collection, we used a program that performed standard Porter stemming [Por80] and stop word removal (the 19 most common words were used as stop words). It sequentially scans all files in the collection, and for each term it encounters writes out a *term* identifier and a *document* identifier, to two separate files. Term identifiers are encoded into a 64-bit number, using a base-27 encoding [Ste09]. Encoding with base 27 allows to represent all the letters

Table 6.2: Global constants used for TREC-TB

Symbol	Meaning
k_1	BM25 parameter (1.2)
b	BM25 parameter (0.5)
n_D	number of documents ($25M$)
$avgdl$	average document length (491)

Table 6.3: SRAM Arrays used for TREC-TB

Name	Symbol	Meaning	$\mathcal{T}$	Sparsity
$LD_{l,d}$	$\exists d_{L,D}$	doc location	([12.3G,25M],bool)	sparse(0)
$LT_{l,t}$	$\exists t_{L,T}$	term location	([12.3G,12M],bool)	sparse(0)
$TD_{t,d}$	$f_{T,D}$	freq of (term,doc)	([12M,25M],int)	sparse(0)
S_d	$\|D\|$	size of doc	([25M],int)	dense
F_t	f_T	doc-freq of term	([12M],int)	dense
Q_t^i	T	term id	([*variable*],lng)	dense

in the English alphabet and results in a maximum term-length of 13 characters. Though not a perfect general-purpose encoding, it proved sufficiently reliable to index English text and efficient thanks to the encoding reversibility, which makes an explicit term dictionary unnecessary.

We can view the resulting single-column files as two-column `[location,term]` and `[location,document]` relations, if we attach to each value a (virtual) densely increasing sequence number `location`. MonetDB/X100 has support for identity columns (`#rownum`), as well as tables stored in binary files, such that these files are treated by it as database relations. These two binary relations, in turn, represent the two sparse boolean matrices location-term $LT_{l,t}$ and location-document $LD_{l,d}$, as used in the Matrix Framework for IR [RTK05]. As mentioned in Section 5.2.1, sparse boolean arrays need not materialise their value column because it is known to be the negated default value. Thus, tables `LD` and `LT` representing sparse boolean matrices $LD_{l,d}$ and $LT_{l,t}$ just consist of their dimension columns (which together form the primary key).

As described in the Matrix Framework for IR, $TD_{t,d}$ is computed in SRAM as a simple matrix multiplication between $LT_{l,t}$ and $LD_{l,d}$ (Line 1 in the code snippet below), S_d as a summation over LD (Line 2) and F_t as a summation over $TD_{t,d}$ (Line 3), whose values are first converted from term-document frequency to term-document presence/absence:

Table 6.4: MonetDB/X100 Relations resulting from Table 6.3

Symbol		Column name	Meaning	Data type	Compression Scheme	Compression Bits
$LD_{l,d}$		LD table – 12.3 Gtuples (output file of parsing)				
l	major-key	LD.location	location id	long	#rownum	0
d	minor-key	LD.docid	document id	int	none	32
$LT_{l,t}$		LT table – 12.3 Gtuples (output file of parsing)				
l	major-key	LT.location	location id	long	#rownum	0
t	minor-key	LT.termid	term id	long	none	64
$TD^*_{t,d}$		TD table – 3.5 Gtuples, term-document info				
t	major-key	TD.termid	term id	long	PFD$_{b=1}$	2.13
d	minor-key	TD.docid	document id	int	PFD$_{b=8}$	11.98
$f_{T,D}$		TD.tf	frequency of T in D	int	PF$_{b=8}$	8.13
$\omega_{T,D}$		TD.score	score of T in D	float	none	32
$\omega'_{T,D}$		TD.scoreQ	quantised score	int	PF$_{b=8}$	8
S_d		D table – 25 Mtuples, document info				
d	key	D.docid	document id	int	#rownum	0
$\|D\|$		D.doclen	document length	int	none	32
F_t		T table – 12 Mtuples, term info				
t	key	T.termid	term id	long	#rownum	0
f_T		T.ftd	doc frequency	int	none	32
$Q^1_t..Q^{50000}_t$		transient Q^i tables – avg. 4 tuples (query len)				
t		Q^i.termid	term id	long	none	64
Compression: PF=PFOR, PFD=PFOR-DELTA, for all base=0						

```
TD := mxMult( mxTrnsp(LT), LD )
S  := [ sum([ LD(l,d) | l ]) | d ]
F  := [ sum([ min( TD(t,d), 1 ) | d ]) | t ]
```

Table 6.4 shows the resulting database schema, with suggestive column names for clarity of presentation.

Since we store the sparse arrays listed above persistently, SRAM makes the corresponding tables clustered on their primary key (i.e. sorted by that order). The most expensive step in data preparation is doing so for $TD_{t,d}$. Sorting its underlying relational table corresponds exactly with creating an inverted list from a table of postings. In all, our Pentium4 server took 7 hours for all TREC-TB data preparation.

As Table 6.4 shows, the full index (the `D`, `T` and `TD(docid, score)` tables) occupies approximately 29GB uncompressed when we ignore the `termid` column in `TD` and replace it with a range index of negligible size. After compressing `TD.docid`, using PFOR-DELTA (see Section 5.4.1), and quantising and compressing `TD.score` using PFOR, the total index size is reduced to roughly 9GB.

6.3.2 TREC-TB experiments

We now report on experiments running the TREC-TB 2005 efficiency task with our IR application on top of SRAM and MonetDB/X100. For these experiments, we used a dual-CPU 3GHz Pentium Xeon (only 1 CPU used for processing) with 4GB of main memory, and a software-RAID system consisting of 12 disks. In all tests we run each query twice, which allows presenting results of *cold* and *hot* runs. The *hot* run represents pure processing time for a query, while the *cold* run gives insight into the overhead of fetching the data from disk. In all experiments presented in this section we used a full Terabyte TREC 2005 dataset and a subset of 5000 randomly chosen queries (with average length of 4 terms). In the following, we describe a number of IR and array optimisation techniques, of which Table 6.5 displays the successive results.

Basic BM25 Query

The array-query in Figure 6.12a, representing BM25 document scoring (see Section 6.3.1), results in the MonetDB/X100 query plan of Figure 6.12b when no optimisation is enabled. Here, the computation of partial scores, as in (6.1), has been replaced with macro `BM25()` for sake of clarity. This physical query plan can be viewed as a logical relational plan by substituting `DenseAggr` for `Aggr`, and `MergeJoin` and `FetchJoin` for `Join` (see e.g. [Gra93] for an overview on database operators and their physical operators).

Since `termid` is the first attribute of the primary key of table `TD` and is ordered, the join with the (tiny) query table `Q` can be handled with a `MergeJoin` (Line 6). This join is quite fast, as it has a sequential access pattern and only touches those disk blocks actually needed, thanks to auxiliary lookup tables with pointers to the first occurrence of `termid` values in `TD`. The fetch-joins perform a foreign-primary key join with an identity column (both `T.termid`, at Line 5, and `D.docid`, at Line 4, are of type `#rownum`) of the small, memory-resident, tables `D` and `T`. This particular kind of join is implemented efficiently in MonetDB/X100 as a pointer-based lookup (see also pointer-based joins in [Gra93]).

Some additional performance analysis showed that of the 1.58 seconds this query takes (see Table 6.5, run BM25) in the hot run, almost 1.35 seconds are spent in the final aggregation. We first experimented with hash-based aggregation. In the TREC-TB queries, hash-based aggregation inserts on average 1.4M documents (from the total 25M) into a hash table, and each aggregate value is processed 3 to 4 times (i.e. as many terms as a query has). The inefficiency of

```
Q = [ 10 42 ]
w(d,t) = log( nDocs / F(t) ) * ((k1 + 1) * TD(t,d))
                               / (TD(t,d) + k1 * ((1 - b) + (b * S(d) / avgdl)))
s(d)   = sum( [ w(d,Q(t)) | t ] )
D20 := topN( [ s(d) | d ], 20, DESC )
```

(a) `BM25.ram`

```
TopN(
  DenseAggr(
    Project(
      FetchJoin(
        FetchJoin(
          MergeJoin(Q, TD, TD.termid == Q.termid),
          T, T.termid == Q.termid),
        D, D.docid = TD.docid),
      [ D.docid, scores = BM25(TD.tf,D.doclen,T.ftd)]),
    [ score = sum(scores) ]),
  [ score DESC ], 20)
```

(b) `BM25.x100`

```
TopN(
  Project(
    MergeOuterJoin(
      RangeSelect( TD1=TD, TD1.termid==10 ),
      RangeSelect( TD2=TD, TD2.termid==42 ),
      TD1.docid = TD2.docid),
    [ S.docid = TD1.docid || TD2.docid, score = TD1.scoreQ + TD2.scoreQ ]),
  [ score DESC ], 20)
```

(c) `BM25UCMQ.x100`

Figure 6.12: The implementation of BM25 (see formulae (6.1) and (6.2)) in our SRAM + MonetDB/X100 system. Figure (a) shows an SRAM 2-term, top-20, query. Figures (b) and (c) show the MonetDB/X100 queries for the basic implementation (BM25) and for the implementation with aggregate unfolding, compression, materialisation and quantisation (BM25UCMQ), respectively.

the hash-based aggregation is explainable as relational implementations of this operator tend to be optimised for the inverse usage pattern (i.e. updating few aggregate values many times). Dense aggregation (`DenseAggr`) can be applied if the GROUP-BY is a single cardinal attribute with a known domain size. In this case, `docid` indeed is a cardinal between 0 and 25M. Internally, dense aggregation creates an in-memory array, used to keep all aggregate totals, that can be accessed by position. Even though the minor ordering of the `TD` table on `docid` will cause "nice" sequential passes over this array for each term in the query, the required initialisation of all 25M floating point aggregate totals to a value of 0.0, and the processing of this 25M result by the top-N operator, still make dense aggregation a costly operator.

Aggregate unfolding

Computing IR scores in queries with many documents and few terms in a query, does not fit well the aggregation algorithms offered by relational database systems (which prefer few often-updated aggregate values over many infrequently updated aggregates). Consider the following SRAM 2-term query, where terms are enumerated explicitly, and used in an aggregate:

```
Q = [ 10 42 ]
s(d) = sum( [ w(d,Q(t)) | t ] )
```

The very low cardinality of array Q_t (i.e. few query terms), together with the availability of explicit query term IDs in the SRAM expression, make the array optimiser consider an alternative plan (see Section 5.1.4), where the summation over the query terms is *unfolded* to a series of additions (here only 1 addition for 2 terms):

```
s(d) = w(d,10) + w(d,42)
```

Because the addition `w(d,10) + w(d,42)` represents a $Map(+, A, B)$ operation between shape-aligned arrays, the corresponding tables can be joined by means of an outer-join operation (see Section 5.2.3). Figure 6.12c shows such a mapping into MonetDB/X100 syntax. Generalising for any query length, the aggregate over the join between `TD` and query table `Q` is transformed into $|Q|-1$ projections over outer-join operations between tuple sequences obtained by selections on table `TD`, based on constant term identifiers as they appear in the explicit array Q. Because table `TD` has `termid` as the major column of its primary key, the RDBMS can use an efficient clustered `RangeSelect` operator per term to obtain such tuple sequences, which avoids the necessity to scan the full column `TD.termid`, by using instead a fast index lookup. This means that the unfolded approach performs less I/O, as the per-query term score-columns layout, together with the column-wise storage of MonetDB/X100 make that only those columns that are used must be read from disk. After such selections based on single `termids`, tuples are still ordered by `docid`, that is the join attribute, which makes it possible to use the implementation `MergeOuterJoin`, optimised for exploiting tuple order.

Apart from allowing such fast sequential access patterns to data, this approach has the additional benefit that it transforms the inefficient sum aggregate (see end of Section 6.3.2) into a series of additions, computed with the `Project` operator. Table 6.5, run BM25U, clearly shows the effect of eliminating the aggregation from the query plan, which accounted for 1.35 seconds both cold and hot: performance improves to 468ms (cold) and 328ms (hot).

The query plan obtained by unfolding the sum aggregate coincides with the well known *Document-At-A-Time* (DAAT) IR processing model, which scores by document, rather than by term as in the *Term-At-A-time* (TAAT) processing model (see [TJ95] for a comparison). The equivalence with the DAAT processing model is completed by the observation that the clustered storage of arrays discussed in Section 5.2.1 makes the $TD_{t,d}$ array act as an inverted list for this processing model. It is interesting to note that the system needs not be explicitly instructed to implement such a strategy. It comes as the result of a more generic rewriting rule, which can be applied to a variety of patterns that do not necessarily find an equivalent in the IR application domain.

Compression

Both in the BM25 and the BM25U runs, we used uncompressed columns. This means that, in case of BM25U, which only touches the `TD.docid` and `TD.tf` columns, we read $32 + 32 = 64$ bits per tuple. If we use the compressed variants of these columns instead (see Table 6.4), the per tuple data size is reduced significantly to $11.98 + 8.13 = 22.11$ bits.

Table 6.5 shows, however, that using the compressed `TD` (run BM25UC) table improves the cold run only marginally. The reason is that the unfolded queries (BM25U as well as BM25UC) are fully computation-bound on the BM25 expression. However, if we estimate the I/O time as the difference between the cold and hot runs, we can still see the benefit of compression (140ms uncompressed versus 106ms compressed). The hot run is only slightly slower than in the uncompressed case, which illustrates that decompression overhead is very minimal. When we use the materialised $TD^{\omega}_{t,d}$, which avoids computation of partial BM25 scores, we see a significant performance enhancement.

One drawback of the materialised representation is that `TD.score` is a 32-bits floating point, which is not easily compressible.

Score materialisation and quantisation

The BM25 score for document D is the sum of all $\omega_{D,T}$ term-document scores for all terms T in the query. The corresponding SRAM macro `w(d,t)` is quite computationally-intensive, as it contains four floating point multiplications and additions, three divisions, and one logarithm. It is a best-practice in IR to precompute (materialise) such partial scores. In principle, SRAM could actually use compiler techniques similar to *loop-invariant code motion* [BGS94], to automatically extract query-independent parts from a scoring formula, and materialise these. We consider this further work. For these experiments, we just declared $w(d,t)$ as a stored array instead of a macro. Run BM25UCM of Table 6.5 shows the large impact of materialisation.

A drawback of materialising floating point $\omega_{D,T}$ scores, is that, unlike the small integer numbers $f_{D,T}$, floating point numbers are much harder to compress (see Section 5.4.1 for more on compression). An alternative to storing the $\omega_{D,T}$ values is to replace these by so-called *score ranks* [AM05a] – small integer values that are the result of quantisation of floating point scores. For example, the *linear Global-By-Value* quantisation

$$\omega'_{D,T} = \left\lfloor q \cdot \frac{\omega_{D,T} - L}{U - L + \eta} \right\rfloor + 1,$$

where L and U are the minimum and maximum values of $\omega_{D,T}$ in the entire collection, produces integer values between 1 and q, and η is a small threshold. Quantisation with too small integers can lead to precision loss in retrieval, and after some tuning experiments we settled for 8-bit integers (bytes). Quantising `TD.score` into the 8-bits `TD.scoreQ` reduces the tuple size to 19.98 bits. This size reduction saves another 33ms in the cold run, bringing it down to 161ms (see Table 6.5, run BM25UCMQ). As expected, quantisation however slightly reduces the precision.

Conjunctive and 2-Pass

The BM25 retrieval model scores each document, regardless the number of matching query terms. Broder et al. [BCH+03] have proposed a 2-pass processing strategy based on the *heuristic* that documents that contain more query terms are likely to obtain a better score. The first pass ranks only documents that contain a set of terms with summed weights exceeding a pre-defined threshold. This may reduce significantly the number of documents considered, improving execution time. Only when the first pass does not return enough documents, a second pass is performed that considers all documents.

We experimented with a simplified version of this approach, where the first pass ranks only documents containing *all* query terms. As this strategy does not compute the true BM25 score, we use the term BM25C for it. The C-suffix stands for “Conjunctive”, as we look for documents that have the conjunction of all query terms (similarly, we may call the real BM25 computation “Disjunctive”). Thus, in TREC-TB, our IR application first computes the top 20 documents using the conjunctive expression, and only if $|D20| < 20$, it actually computes the top-N of the disjunctive expression. If a query has fewer terms, if these terms are more frequent, or if these are likely to co-occur, the probability of the conjunctive expression finding enough documents increases. In the TREC-TB query set, the second pass turns out to be necessary in only 15% of the queries.

The conjunctive scoring formula multiplies the BM25 score with a (0,1) boolean value that is only 1 for those documents that have all terms:

$$S^{(D)}_{BM25C} = \prod_{T \in Q} (\omega_{D,T} > 0) * \sum_{T \in Q} \omega_{D,T}$$

This is transcribed in SRAM syntax as follows:

```
s(d) =  prod( [ w(d,Q(t)) > 0 | t ] )
      *  sum( [ w(d,Q(t))     | t ] )
```

Multiplications of arrays with such sparse (boolean) matrices can significantly improve the running time of a query. The underlying reason is that such multiplications can make the result sparser, and thus smaller, when considering its representation as a relational table. This is why the conjunctive variant, being a multiplication on the disjunctive (normal) BM25 formula, turns out to be much faster, and makes the 2-pass strategy beneficial (see Section 6.3.2).

Although this expression adds computations to the one for the normal BM25 query, we describe here how query optimisation makes this formula much faster to compute (all the optimisation techniques used here are described in Chapter 5). By unfolding both aggregates, as explained in the previous section, the expression above is rewritten into:

```
s(d) = (w(d,t1) > 0) * (w(d,t2) > 0) * ...
     * ( w(d,t1) + w(d,t2) + ... )
```

The arithmetic optimisation (see Section 5.2.2) and rules 4.7, 5.21 and 4.2 translate the multiplications in the expression above into projections on top of join operators. In contrast, for the additions in the second line, according to Rule 5.24, full outer-join operators are needed.

If W^{tj} denotes the relation corresponding to each partial score `w(d,tj)`, the join sequence for a 3-term expression becomes as follows (ignoring projections):

$$\begin{tikzcd}
 & & & \bowtie_d \arrow[ll, dash] \arrow[rr, dash] & & & \\
 & \bowtie_d \arrow[dl, dash] \arrow[dr, dash] & & & & {=}\bowtie{=}_d \arrow[dl, dash] \arrow[dr, dash] & \\
\bowtie_d \arrow[d, dash] \arrow[dr, dash] & & W^{t3} & & {=}\bowtie{=}_d \arrow[d, dash] \arrow[dr, dash] & & W^{t3} \\
W^{t1} & W^{t2} & & & W^{t1} & W^{t2} &
\end{tikzcd}$$

All join and outer-join equality conditions are on attribute d, which is the only dimension of the corresponding array, thus the key attributes for all the relations W^{tj}. Following the optimisation techniques introduced in [GLR97], this allows the simplification of outer-join operators into join operators, as all the `NULL`-padded tuples produced by outer-joins will be subsequently discarded by the joins on the same key:

$$\begin{tikzcd}
 & & & \bowtie_d \arrow[ll, dash] \arrow[rr, dash] & & & \\
 & \bowtie_d \arrow[dl, dash] \arrow[dr, dash] & & & & \bowtie_d \arrow[dl, dash] \arrow[dr, dash] & \\
\bowtie_d \arrow[d, dash] \arrow[dr, dash] & & W^{t3} & & \bowtie_d \arrow[d, dash] \arrow[dr, dash] & & W^{t3} \\
W^{t1} & W^{t2} & & & W^{t1} & W^{t2} &
\end{tikzcd}$$

The same equality conditions allow to verify that the expression above contains redundant key equi-join operations. Standard join elimination techniques [CGM90] reduce it to the following expression:

$$\begin{tikzcd}
 & \bowtie_d \arrow[dl, dash] \arrow[dr, dash] & \\
\bowtie_d \arrow[d, dash] \arrow[dr, dash] & & W^{t3} \\
W^{t1} & W^{t2} &
\end{tikzcd}$$

Thus, the MonetDB/X100 query plan for the conjunctive version of BM25, named BM25UCMQC, is equivalent to the disjunctive version in Figure 6.12c, except for the `MergeJoin` operator that has replaced the `MergeOuterJoin` operator. Finally, standard join order optimisation [SMK93] can schedule the execution of the most selective join operations first, to reduce as soon as possible the cardinality of intermediate results. In IR words, this means computing the score of the least frequent terms first, exploiting the typical skew of the Zipfian term distribution to discard document candidates early, as done in [BL85] in the context of an upper-bound search algorithm.

Table 6.5: SRAM and MonetDB/X100 on TREC-TB

Run name (+ added feature)	p@20	Avg (ms)	
		Cold	Hot
12-disk 3GHz Pentium4 4GB RAM server			
BM25	0.546	1826	1584
BM25U (+ Unfolding)	0.546	468	328
BM25UC (+ Compression)	0.546	439	333
BM25UCM (+ Materialisation)	0.546	194	65
BM25UCMQ (+ Quantisation)	0.543	161	63
BM25UCMQC (+ Conjunctive)	0.538	109	13
BM25UCMQC2 (+ 2-Pass)	0.547	117	21

Replacing the outer-joins with joins and re-ordering them by cardinality (i.e. term frequency) has as effect that the number of candidate documents quickly decreases. Whereas the top-N in the default plan chooses among 1.4M documents on average, in the conjunctive case the average amount of candidates is down to 62K, such that the query executes much quicker in the hot run (from 63ms to 13ms). The number of candidates still varies widely, and in 15% of the TREC-TB queries, the conjunctive strategy yields in fact *less* than the 20 documents required for P@20. This explains the deterioration in precision to 0.538, visible in Table 6.5, run BM25UCMQC. This deterioration is mitigated using the 2-pass strategy, settling our cold run at 117ms and the hot run at 21ms (run BM25UCMQC2).

6.3.3 Discussion

To put our results in perspective, Table 6.6 summarises the efficiency and precision of three custom IR engines that were made available for comparative runs in the 2006 TeraByte Track. These runs were performed on the same hardware as our single server-based runs, using the same 2005 efficiency topics, and operating on cold data. This makes for a fair comparison of the various systems.

Table 6.6 illustrates that the results of SRAM + MonetDB/X100 are competitive, second only to Wumpus in terms of efficiency (117ms vs. 91 ms on cold data). In terms of precision, SRAM + MonetDB/X100 scores second as well, after Indri (0.561 vs. 0.547). However, Indri's win in terms of precision comes at a significant cost in terms of execution time (117ms vs. 1724 ms).

The following sections highlight some of the conclusions we were able to draw from the experiments on TREC-TB. In this context, the array data model turned

Table 6.6: Performance compared to custom IR systems (cold runs)

System	Index size (GB)	p@20	CPUs	Time per query (ms)
Indri	100	0.5610	1	1724
Zettair	44	0.4770	1	390
Wumpus	14	0.5310	1	91
SRAM-MonetDB/X100	9	0.5470	1	117

out to be well-suited for describing IR tasks not only theoretically (as in the Matrix Framework for IR), but also in practice when properly mapped onto the relational domain.

Next-generation databases are good for IR. A first way to interpret these results is as a proof that there is a misconception around the belief of database technology not being able to provide raw performance that is competitive with that of custom-built IR solutions. The research on MonetDB/X100 advocates that the engineering of existing database engines did not keep up with recent hardware's evolution and that next-generation execution models are necessary to exploit the advantages of modern hardware, following the example of other application areas (e.g. computer graphics, scientific tools, computer games, see [ABH09]). The results presented in this section show that, from a performance point of view, exploitation of the computational power offered by modern hardware by database engines is indeed a key missing ingredient in the so far failed attempts for DB and IR integration (see [CRW05, Wei07]).

IR and array optimisations mapped onto relational optimisations. The second necessary ingredient for high-performance DB-powered IR, together with a next-generation database engine, is the ability to feed this engine with proper inputs, that is, to generate optimised physical query plans for the declarative problem specification at hand. In the past, this has not even succeeded when starting from SQL, even though SQL is much closer to the database domain than to the IR domain. A second achievement of the experiments presented in this section is to show how database query plans suited for IR can be obtained starting from the array domain, which is then confirmed to represent a well-suited abstraction between IR and DB domains (after the work in [RTK05] showed it is a well-suited abstraction for IR alone).

The relational DAAT query plans that perform inverted list merging, used here to achieve our highest performance results, are not new at all, but we do consider it a major success that we were able to generate these plans automatically, basically starting from the mathematical formula that defines BM25 and with no IR-specific array mapping rule. Experiments on large collections such as TREC-TB demonstrate that the array paradigm proposed in the Matrix Framework for IR can only be implemented using *sparse* arrays (dense materialisation arrays like LT or TD is simply unfeasible on non-trivial collections). The results confirm that the effort put in mapping sparse array computations onto efficient relational queries is an approach that is practically viable, resulting in performance competitive with that of customised IR systems.

One database-powered framework for many retrieval models. The same collection index based on $LD_{l,d}$ and $LT_{l,t}$ matrices can be used to implement other retrieval models as described in [RTK05]. We briefly discuss the Language Modelling approach [Hie98, PC98]:

$$S_{LM}^{(D)} = \sum_{T \in Q} \omega_{D,T} \tag{6.3}$$

$$\omega_{D,T} = \log\left(\lambda \cdot P(T|D) + (1-\lambda) \cdot P(T)\right) \tag{6.4}$$

$$\propto \log\left(\frac{\lambda \cdot P(T|D)}{(1-\lambda) \cdot P(T)} + 1\right) \tag{6.5}$$

$$P(T|D) = \frac{f_{T,D}}{|D|}, \qquad P(T) = \frac{fc_T}{n_L}$$

where fc_T and n_L denote collection term-frequency and collection size (number of locations), respectively.

Formulae (6.4) and (6.5) produce equivalent rankings. The latter uses a presence (as opposed to presence/absence) weighting scheme [RSJ88] and can be implemented more efficiently, as it assigns a zero weight to terms that are not present in a document. In sparse matrix operations, this means significantly less computation.

The following SRAM expression implements Language Modelling with presence weighting scheme:

```
s(d)     = sum([ w(d,Q(t)) | t ])
w(d,t)   = log( l * ptd(t,d)
                / (1 - l) * pt(t)) + 1 )
ptd(t,d) = TD(t,d) / S(d)
pt(t)    = Fc(t) / nLocs
```

with the additional array fc_T created at indexing time:

```
Fc := [ sum([ LT(l,t) | l ]) | t ]
```

An interesting area for future research includes automatic mathematical reformulations such as the one from (6.4) to (6.5). The system would be allowed to consider the two versions as equivalent when a query asks for a ranking, with the `topN` syntax, but not for the actual scores.

Last words are for fools
who haven't said enough
- Karl Marx, economist

7 Conclusions

The research presented in this thesis, summarised in Section 7.1, proposes a structured approach to the engineering of search systems. A clean separation of concerns is realised through the identification of abstraction layers, which provide the correct amount of declarativeness at each stage in the search process. This layered architecture, together with suitable instantiations of the abstraction layers identified and carefully deigned mapping strategies, allows to enable content independence (search strategies independent of content representation) and data independence (data access strategies independent of data representation). As experimentally shown, although this high degree of flexibility can be obtained with a fully automated translation and optimisation process, efficiency and scalability can compete with highly optimised custom-built search solutions.

7.1 Contributions

Assuming a layered architecture, which abstraction layers should be distinguished, as a prerequisite for providing search systems with content and data independence?

Chapter 3 introduces the concept of a Parametrised Search System (PSS). Distinguishing feature of a PSS is a layered architecture that facilitates the decoupling of search strategies from logical content representation, algorithms and physical organisation of data, which is the pre-requisite for enabling content and data independence in search systems. The abstraction layers identified in Chapter 3 for a generic PSS are:

application interface - to express the user information need
(e.g. a simple keyword-list, a query-by-example interface, etc.)

application abstraction - to model and abstract the information need (IR details are abstracted away)

IR modelling abstraction - to model an abstract IR task (data-access details are abstracted away)

conceptual data-access layer - to express an IR task declaratively (actual expression of a modelled IR task, logical/physical data-access details are abstracted away)

logical data-access layer - to express a data-access plan declaratively (instantiation of a conceptual data-access abstraction, physical details are abstracted away)

physical data-access layer - to express a physical data-access plan.

The first two abstraction layers depend on the specific application-domain and do not play a major role in providing the pursued architectural flexibility. **Content independence** is enabled by choosing suitable instantiations for the IR modelling abstraction and the conceptual data-access abstraction. **Data independence** is enabled by mapping the conceptual data-access abstraction chosen onto suitable logical and physical data-access abstractions. As the results in Chapter 6 show, content and data independence can be achieved in a PSS because of the clean separation of concerns: this allows to index and search different content (e.g. XML, flat text, graphs) without modifying the software, but describing data, indexing and search strategies declaratively.

What are suitable instantiations of such abstraction layers?

Leaving the first two application-specific abstraction layers unspecified, as they are not expected to influence the PSS flexibility, this thesis proposes to instantiate each remaining abstraction layer as follows:

IR modelling abstraction: a framework based on matrix spaces, as the Matrix Framework for IR

conceptual data-access layer: the array data model

logical data-access layer: the relational algebra

physical data-access layer: a database engine

Chapter 6 applies these concepts to a real PSS implementation. The implementation experiments in sections 6.1, 6.2 and 6.3 show that modelling IR problems in terms of matrix spaces is independent of the specific content representation (XML, graphs and flat text respectively). This is in line with what claimed in [RTK05]. Additionally, we have shown feasibility of such a content-independent modelling by mapping the theoretical matrix framework onto a real implementation based on the array data-model.

Data independence is well-known as one of the crucial benefits provided by database technology. This reason, together with the amount of solutions for data retrieval and database query optimisation, assumed relational algebra and database technology the preferred instantiation of the logical and physical data-access abstraction layers. Chapter 6 shows implementation of the complete stack of layers.

Can automatic mappings among the identified layers be defined, so that possible data-model mismatches are taken care of by the system?

Identifying reasonable instantiations for each abstraction layer in isolation, so that it plays well its role, is only part of the challenge. Another important aspect is to make sure that such abstraction layers work well together, in a pipeline that smoothly brings the required information from the underlying raw data up to the user interface. Some abstractions are very close to each other, such as a framework based on a matrix-space and a data-access abstraction based on the array data-model. The next step, mapping the array data-model onto relational algebra, is less trivial. This is the topic discussed in Chapter 4, where a complete mapping framework is presented for the array-syntax proposed. All experiments in Chapter 6 show that when most common IR problem are modelled as in the Matrix Framework for IR and expressed in SRAM syntax, these can be mapped successfully onto relational algebra and physical query plans.

What are the requirements of such a software architecture in terms of runtime efficiency and scalability?

This topic is addressed in Chapter 5 and evaluated mainly in Section 6.3. We can draw a number of conclusions:

- Mappings from one domain to another should include expressions as well as optimisations. The array-to-relation mapping process is supported by a set of specific optimisation rules, that operate in three distinct phases: in the original domain, during the mapping itself and in the target domain. Implementing the needed optimisations at all levels can enable all-round and transparent optimisations, as exemplified in Section 6.3.2, where the typical IR processing based on inverted-file structure is mimicked in the relational world, thanks to generic optimisation techniques, or in Section 6.3.2, where a mathematically-defined IR optimisation has been propagated and translated into relational optimisations, without any specific customisation.

- Database engines have been described in past experiments as too slow for the computation-intensive processing typical of IR tasks [CRW05]. Section 5.4.1 advocates that this statement, as it is, should be considered ill-posed nowadays, as it starts from an outdated conception of database technology. Not only we could show for the first time how a new-generation, hardware-conscious database architecture can provide IR applications with efficient data access (see Section 6.3). We have also shown that this competitive performance can be obtained through automatic query translation, starting from declarative problem specifications in the array domain.

7.2 Future work

This section summarises some challenges that this thesis leaves open (or opens up) and that should be considered as interesting future work.

Increased awareness of IR data distributions. The search system powered by SRAM and MonetDB/X100 technologies proved highly efficient for large-scale retrieval tasks. However, neither technology is specifically optimised for the typical Zipfian distribution of data in IR corpora. The standard implementation of hash-aggregation in database engines is one of the crucial non-IR friendly DB technologies. When, for example, a per-group summation can be computed on a set of values, groups of tuples must be identified. This is an easy problem when tuples are already sorted on the grouping-attributes. Otherwise, a common way to identify groups is to construct a hash table, where buckets keep the current aggregated value for each group (see [Gra93]). One of the key factors that determine the efficiency of hash-aggregation is the number of buckets of the hash table, which depends on the expected number (or average size) of the groups in the input data. Using too few buckets results in many groups per bucket and consequently many lookup steps needed per tuple; using too many buckets consumes more memory, with possible consequences on the overall performance of the system. Typically, the correct hash table size is estimated by reading small samples of input data. This works reasonably well on uniformly distributed data, but is bound to produce bad estimations on highly skewed data. In the IR domain this happens when, for example, a per-document term frequency needs to be computed from a binary (presence/absence) term-document relation, using a `count` aggregation function. Because of the Zipfian distribution of natural language terms, the size of term-doc groups can vary greatly: few terms occur very frequently, whereas the majority of terms occur only few times. A more effective way of solving the problem would properly propagate detailed information from domain-specific corpora down to the relational execution engine. How to collect such knowledge, propagate it through all the architecture layers and exploit it at best in general-purpose data-access algorithms is a challenging topic to be explored.

More flexible and efficient relational representations. The relational storage scheme proposed in Section 4.5.1, in combination with array-to-relational mapping and optimisation rules (see chapters 4 and 5), enables a powerful and flexible engine for array computations (see Chapter 6). However, a large amount of literature on array storage and computation supports the argument that a "one size fits all" storage scheme is not likely to provide overall efficient array processing, especially when dealing with sparse arrays. For example, Section 6.1.3 shows how the default SRAM storage scheme can be sub-optimal when used to encode hierarchically structured data as XML, where range-based encodings may be more suited, following the example of MonetDB/XQuery [BGvK+06]. Also, sections 6.2.6, 6.2.7 and 6.3.1 make explicit that efficient physical query plans depend on the chosen storage properties for each array. Within the same storage scheme, the choice of which is the major axis (and which is the importance order of the remaining axes) translates into the choice of which index column an array table is sorted on. This triggers the choice of physical operators to use (e.g. hash-join instead of a faster merge-join, when join columns are not sorted). Because it is not possible to have the same table sorted on different attributes, the same tables are bound to be accessed with different efficiency by different queries. One possible solution would be to support replication of tables into a number of copies optimised for different access patterns. Ability to support and integrate diverse storage schemes with an efficient array-to-relational mapping process is of primary importance in the future development of this research (see [CMWM09] for a closely relate work on this topic).

Support for iterative computations. The random walk on a social network (Section 6.2) is an iterative retrieval task. Because this is based on a fixed and relatively small number of iterations, the limited support for loops provided by the

SRAM system (see Section 4.3.1) is sufficient. However, a number of tasks that are common in IR require a high and non-fixed (i.e. data-dependent) number of iterations. Examples include clustering algorithms used in unsupervisioned learning, such as k-means and expectation maximisation (EM). Support for iterations in the array domain that overcomes the limitations imposed by the SRAM syntax is certainly possible. Also, the internals of some relational execution engines provide rich data-interfaces that include proper support for loops (e.g. [CWI]). However, relational algebra completely lacks the concept of iteration, if not explicitly unrolled and encoded as data-replication. While replication is possible using cartesian products, it is inefficient and does not support data-dependent loop-exit conditions. Extending the relational algebra layer with iterations would bridge the gap and enable relational-powered, iterative array computations.

A
SRAM syntax normalisation rules

This appendix presents details about the SRAM syntax normalisation process that is described informally in Section 4.4.1. The process consists of two stages: flattening and shape alignment, indicated with $\Phi(\mathtt{E})$ and $\Upsilon_\varsigma(\mathtt{E})$, respectively. In practice, the normalisation process is a top-down application of translation rules dealing with the different constructs of comprehension syntax. First, the original expression is flattened. Then, the flattened expression is recursively aligned to the shape of the result array:

$$\begin{aligned} &\mathtt{E}' = \Phi(\mathtt{E}) \\ &\mathtt{E}'' = \Upsilon_\varsigma(\mathtt{E}'), \quad \varsigma = (\psi(\mathtt{E}')\mathtt{<}\mathcal{S}_{\mathtt{E}'}) \end{aligned}$$

Table A.1 summarises the notation used in this appendix.

A.1 Flattening

Recall from Definition 4.4 that array elements can be arrays themselves, which yields nested array structures. The flattening step normalises arrays to their non-nested equivalent. The basic principle is that a nested array E can always be represented as a flat array E' whose valence is the sum of the valences of the array structures nested in E.

The set of rules below, which are matched in the presented order against input expressions, are applied repeatedly to the top of the expression tree, until this cannot be further transformed:

Table A.1: Notation used in normalisation rules (see Table 4.1 at page 33 for other symbols used)

Notation	Meaning
`E`	A generic array expression in comprehension syntax
Φ(`E`)	Flatten array expression `E`
Υ_ς(`E`)	Align shape of array expression `E` to ς
ς	($\bar{\imath}$<$\mathcal{S}$), axis identifiers and shape to be used for reshaping an array expression
ψ(`E`)	Vector of axis identifiers defined for `E`

```
repeat
  E_old = E
  E = $\Phi$(E_old)
until E $=$ E_old

$\Phi$(E) = {
  E = Rule A.1 (E)
  E = Rule A.2 (E)
  ...
}
```

This is necessary because some of the rules presented below may generate nested sub-expressions as a side-effect.

Only Rule A.1 actually removes nesting layers. All the other rules either make implicit nestings explicit, by translating them into comprehensions, or merely propagating the flattening operator throughout the different expression patterns.

Rule A.1 (Flattening nested comprehensions). This is the only rule that actually performs unnesting. The outer comprehension is removed and its index domain pre-pended to the inner comprehension's index domain. The result is a higher-dimensionality, flat array:

$$\frac{\Phi(\texttt{[[E | } \bar{e}\texttt{<}\mathcal{S}' \texttt{] | } \bar{\imath}\texttt{<}\mathcal{S} \texttt{]})}{\Phi(\texttt{[E | } \bar{\imath}\texttt{<}\mathcal{S}\texttt{, } \bar{e}\texttt{<}\mathcal{S}' \texttt{]})}$$

Once removed the most "external layer of the onion", the process is applied again to the result. □

Rule A.2 (Flattening comprehensions). When the expression inside a comprehension is not a comprehension itself, the flattening process is applied to it.

$$\frac{\Phi(\texttt{[E | } \bar{\imath}\texttt{<}\mathcal{S} \texttt{]})}{\texttt{[}\Phi(\texttt{E})\texttt{ | } \bar{\imath}\texttt{<}\mathcal{S} \texttt{]}}$$

Note that this, depending on the syntactic pattern of the expression, may generate new comprehensions. This requires a new flattening process to take place at the syntactic tree root. □

Rule A.3 (Flattening nested expressions). When a generic expression E is nested (recall Definition 4.7), every element $\mathtt{E}(\psi(\mathtt{E}))$ is an array itself. This rule transforms expression E into a comprehension, which makes array elements explicit. The comprehension just created can later be removed by Rule A.1 and the array elements flattened recursively.

$$\frac{\Phi(\mathtt{E}) \qquad \forall \bar{\imath} \in \mathcal{D}_E : |\mathcal{S}_{E(\bar{\imath})}| > 0}{\mathtt{[}\ \Phi(\mathtt{E}(\psi(\mathtt{E})))\ \mathtt{|}\ \psi(\mathtt{E})\mathtt{<}\mathcal{S}_E\ \mathtt{]}}$$

Intuitively, this rule takes a step deeper into the recursive structure of nested arrays, from an upper-level into its elements. □

Rule A.4 (Flattening applications of persistent arrays). This rule is one of the termination points of the flattening algorithm. Applications of persistent arrays to a set of index vectors cannot be further flattened when they yield an atomic value. This rule applies to both nested and flat arrays.

$$\frac{\Phi(\mathtt{A}(\bar{\imath})\cdots(\bar{\jmath})) \qquad |\mathcal{S}_{A(\bar{\imath})\cdots(\bar{\jmath})}| = 0}{\mathtt{A}(\bar{\imath})\cdots(\bar{\jmath})}$$

□

Rule A.5 (Flattening applications). Applications that do not match the patterns described in Rules A.3 and A.4 deal with possibly nested non-persistent arrays that are applied to a set of index vectors. Because the array is non-persistent, it can be recursively flattened and matched with the flat concatenation of its index vectors.

$$\frac{\Phi(\mathtt{E}(\bar{\imath})\cdots(\bar{\jmath}))}{\Phi(\mathtt{E})(\bar{\imath},\ldots,\bar{\jmath})}$$

□

Rule A.6 (Flattening functions). Because functions are defined over atomic values, their arguments are guaranteed not to be arrays. However, sub-expressions of these may result in intermediate nested arrays, which need to be flattened by propagation of the flattening operator.

$$\frac{\Phi(\mathtt{f(E_1,\ldots,E_m)})}{\mathtt{f(}\Phi(\mathtt{E_1}),\ldots,\Phi(\mathtt{E_m})\mathtt{)}} \qquad \square$$

Rule A.7 (Flattening aggregations). Aggregation functions are defined from arrays to atomic values. The flattening operator is propagated to their argument to ensure that the result is indeed an atomic value.

$$\frac{\Phi(\mathtt{f(E)})}{\mathtt{f(}\Phi(\mathtt{E})\mathtt{)}} \qquad \square$$

Rule A.8 (Flattening sort and topN constructs). Sub-expressions of Sort and topN operators may result in intermediate nested arrays, which need to be flattened by propagation of the flattening operator.

$$\frac{\Phi(\mathtt{sort(E,d)})}{\mathtt{sort(}\Phi(\mathtt{E})\mathtt{,d)}} \quad \frac{\Phi(\mathtt{topN(E,n,d)})}{\mathtt{topN(}\Phi(\mathtt{E})\mathtt{,n,d)}} \qquad \square$$

Rule A.9 (Flattening anything else). The most generic termination rule for the flattening algorithm deals with all expressions that do not match any of the previous patterns, such as atomic values, axis identifiers or flat arrays.

$$\frac{\Phi(\mathtt{E})}{\mathtt{E}} \qquad \square$$

Example A.1 (Flattening a nested array).

```
                A = ([10,5],([20],int)) "A"
                Φ([ A(x,y) | y, x ])

Rule A.2 ⟶     [ Φ(A(x,y)) | y, x ]
Rule A.3 ⟶     [ [ Φ(A(x,y)(z)) | z ] | y, x ]
Rule A.4 ⟶     [ [ A(x,y)(z) | z ] y, x ]
2nd pass ⟶     Φ([ [ A(x,y)(z) | z ] y, x ])
Rule A.1 ⟶     [ A(x,y)(z) | y, x, z ]
```

Example A.2 (Flattening a nested comprehension).

```
                A = ([10,5],int) "A"
                Φ([ [ [ A(x,y) | x,y ] | z ](z) | z ])

Rule A.2 ⟶     [ Φ([ [ A(x,y) | x,y ] | z ](z)) | z ]
Rule A.3 ⟶     [ [ Φ([ [ A(x,y) | x,y ] | z ](z)(x,y)) | x,y ] | z ]
Rule A.5 ⟶     [ [ Φ([ [ A(x,y) | x,y ] | z ])(z,x,y) | x,y ] | z ]
Rule A.1 ⟶     [ [ Φ([ A(x,y) | z,x,y ])(z,x,y) | x,y ] | z ]
Rule A.2 ⟶     [ [ [ Φ(A(x,y)) | z,x,y ](z,x,y) | x,y ] | z ]
Rule A.4 ⟶     [ [ [ A(x,y) | z,x,y ](z,x,y) | x,y ] | z ]
2nd pass        Φ([ [ [ A(x,y) | z,x,y ](z,x,y) | x,y ] | z ])
Rule A.1 ⟶     [ [ A(x,y) | z,x,y ](z,x,y) | z,x,y ]
```

A.2 Shape alignment

This normalisation process, which takes place on a previously flattened expression, helps the recognition of patterns that match the definition of array algebra operators (see Section 4.4.1). It works by reshaping all sub-expressions to match the shape of the expression's result, which is denoted in the following rules, together with its axes, as $\varsigma = (\bar{\imath}{<}\mathcal{S})$. Note that these rules are not robust against incorrect or ill-defined expressions, such as `[ [ f(x,y) | x,y ](z) | z ]`, where a 2-dimensional array is applied to one array index (`z`). The rules below are matched in the presented order against input expressions. Input expression correctness must be checked before normalisation takes place.

Rule A.10 (Aligning comprehensions). Align comprehensions to a new shape simply involves replacing the existing shape specification with a new one.

$$\frac{\Upsilon_\varsigma(\texttt{[E | }\bar{e}{<}\mathcal{S}'\texttt{]}) \qquad \varsigma = (\bar{\imath}{<}\mathcal{S})}{\texttt{[}\Upsilon_\varsigma(\texttt{E}(\bar{\imath}))\texttt{ | }\bar{\imath}{<}\mathcal{S}\texttt{]}}$$

It is assumed that `E(`$\bar{\imath}$`)` is well-defined, i.e. $\psi(\texttt{E}) \subset \bar{\imath}$. □

Rule A.11 (Aligning persistent arrays). In order to align persistent arrays, these need to be first rewritten into an explicit comprehension expression, which can then be aligned to a new shape by replacing the existing shape specification.

$$\frac{\Upsilon_{\varsigma}(\texttt{A}) \qquad \varsigma = (\bar{\imath}\texttt{<}\mathcal{S})}{\texttt{[A(}\psi(\texttt{A})\texttt{) | }\bar{\imath}\texttt{<}\mathcal{S}\texttt{]}}$$

It is assumed that $\psi(\texttt{A}) \subset \bar{\imath}$. □

Rule A.12 (Aligning applications of persistent arrays). Alignment of persistent array applications to sets of indices cannot be further propagated. An explicit comprehension is created around the original expression, which often yields an identity transformation.

$$\frac{\Upsilon_{\varsigma}(\texttt{A(}\bar{\jmath}\texttt{)}) \qquad \varsigma = (\bar{\imath}\texttt{<}\mathcal{S}) \qquad |\mathcal{S}_{A(\bar{\jmath})}| = 0}{\texttt{[A(}\bar{\jmath}\texttt{) | }\bar{\imath}\texttt{<}\mathcal{S}\texttt{](}\bar{\imath}\texttt{)}}$$

It is assumed that $\bar{\jmath} \subset \bar{\imath}$. □

Rule A.13 (Aligning applications). Alignment of array expression applications to a set of indices can be recursively propagated to both parts of the original expression.

$$\frac{\Upsilon_{\varsigma}(\texttt{E(}\bar{\jmath}\texttt{)}) \qquad \varsigma = (\bar{\imath}\texttt{<}\mathcal{S})}{\begin{gathered}\Upsilon_{\varsigma'}(\texttt{E})\texttt{(}\Upsilon_{\varsigma}(\bar{\imath})\texttt{)} \\ \varsigma' = \bar{\imath}'\texttt{<}\mathcal{S} \\ \forall \bar{\imath}'_l \in \bar{\imath}' : \bar{\imath}'_l = \begin{cases} \psi(\texttt{E})_k & \text{if } \exists k : j_k = i_l \\ i_l & \text{otherwise} \end{cases}\end{gathered}}$$

Because expression `E` is guaranteed to be aligned to the new shape $\mathcal{S}$, the axis identifiers $\bar{\jmath}$ need to be replaced with $\bar{\imath}$, in order to match the new shape. However, when aligning recursively expression `E`, axis identifiers in ς need to be renamed so that they match those available inside `E`. The example below shows this renaming. □

Rule A.14 (Aligning aggregations). Aggregation is perhaps the least straightforward construct in array comprehension syntax. Recall that, in `f([ E | `$\bar{\jmath}$`<`$\mathcal{S}'$` ])`, expression $\bar{\jmath}$`<`$\mathcal{S}'$ does not identify the comprehension's shape, but rather the axes that are to be collapsed by the aggregation in (therefore lost from) expression `E`. The shape alignment operation can be propagated as normal, except that the shape being propagated must include those axes $\bar{\jmath}$ that are being discarded by the aggregation. The final result will not include them, but the argument of the aggregation does.

Moreover, this rule performs an additional transformation that is not strictly concerned with shape alignment. It rewrites the aggregation function into an `aggr(f,n,E)` construct that facilitates the transformation into an array algebra operator, by shifting to the end the `n` axes that are to be collapsed. See the transformation rule to the *Aggregate* array algebra operator, in Section 4.4.2, for further details.

$$\frac{\Upsilon_{\varsigma}(\texttt{f([E | }\bar{\jmath}\texttt{<}\mathcal{S}'\texttt{]))} \qquad \varsigma = (\bar{\imath}\texttt{<}\mathcal{S})}{\begin{array}{c}\texttt{aggr(f, }|\bar{\jmath}|\texttt{, }\Upsilon_{\varsigma'}\texttt{(E))} \\ \varsigma' = (\bar{\imath} \mathbin{++} \bar{\jmath})\texttt{<}(\mathcal{S} \mathbin{++} \mathcal{S}')\end{array}} \qquad \square$$

Rule A.15 (Aligning functions). Shape alignment is simply propagated from functions to their arguments.

$$\frac{\Upsilon_{\varsigma}(\texttt{f(E}_1\texttt{, ..., E}_m\texttt{))}}{\texttt{f(}\Upsilon_{\varsigma}(\texttt{E}_1)\texttt{,...,}\Upsilon_{\varsigma}(\texttt{E}_m)\texttt{)}} \qquad \square$$

Rule A.16 (Aligning axis vectors). Aligning axis vectors is translated into alignment of each axis in that vector.

$$\frac{\begin{array}{c}\Upsilon_{\varsigma}(\bar{\jmath}) \qquad \varsigma = (\bar{\imath}\texttt{<}\mathcal{S}) \\ \bar{\jmath} = \texttt{(}j_1\texttt{, ..., }j_k\texttt{)}\end{array}}{\texttt{(}\Upsilon_{\varsigma}(j_1)\texttt{,...,}\Upsilon_{\varsigma}(j_k)\texttt{)}} \qquad \square$$

Rule A.17 (Aligning axes). A single axis identifier is aligned to a shape by creating a comprehension of the given shape and selecting the enumeration of given axis as an array value. This corresponds to creating an array index, that is, an array whose values are index values.

$$\frac{\begin{array}{c}\Upsilon_{\varsigma}(\texttt{j}) \qquad \varsigma = (\bar{\imath}\texttt{<}\mathcal{S}) \\ \texttt{j} \in \bar{\imath}\end{array}}{\texttt{[j | }\bar{\imath}\texttt{<}\mathcal{S}\texttt{](}\bar{\imath}\texttt{)}}$$

□

Rule A.18 (Aligning constants). A constant value is aligned to a shape by creating a comprehension of the given shape and selecting given constant as an array value. This corresponds to creating constant array.

$$\frac{\Upsilon_{\varsigma}(c) \qquad \varsigma = (\bar{\imath}\texttt{<}\mathcal{S})}{\texttt{[}c\texttt{ | }\bar{\imath}\texttt{<}\mathcal{S}\texttt{](}\bar{\imath}\texttt{)}}$$

□

Rule A.19 (Aligning sort and topN constructs). Sort and topN operators, like functions, simply propagate the flattening operator to their arguments.

$$\frac{\Upsilon_{\varsigma}(\texttt{sort(E,d)})}{\texttt{sort(}\Upsilon_{\varsigma}(\texttt{E})\texttt{,d)}} \qquad \frac{\Upsilon_{\varsigma}(\texttt{topN(E,n,d)})}{\texttt{topN(}\Upsilon_{\varsigma}(\texttt{E})\texttt{,n,d)}}$$

□

Example A.3 (Aligning a flattened comprehension).

$$\begin{array}{ll}
 & \texttt{A = ([10,5],int) "A"} \\
 & \Upsilon_{\varsigma}(\texttt{[A(x,x) | x](z) | z<4,w<20)}, \quad \varsigma = \texttt{z<4,w<20} \\
\text{Rule A.13} \longrightarrow & \texttt{[}\Upsilon_{\varsigma'}(\texttt{[A(x,x) | x]})(\Upsilon_{\varsigma}(\texttt{z}))\texttt{ | z<4,w<20]} \\
 & \varsigma' = \texttt{x<4,w<20}, \varsigma = \texttt{z<4,w<20} \\
\text{Rule A.10} \longrightarrow & \texttt{[[}\Upsilon_{\varsigma'}(\texttt{A(x,x)})\texttt{ | x<4,w<20]([z | z<4,w<20](z,w)) | z<4,w<20]} \\
 & \varsigma' = \texttt{x<4,w<20} \\
\text{Rule A.12} \longrightarrow & \texttt{[[[A(x,x) | x<4,w<20](x,w) | x<4, w<20]([z | z<4,w<20](z,w)) | z<4,w<20]}
\end{array}$$

Bibliography

[AAB$^+$05] Serge Abiteboul, Rakesh Agrawal, Phil Bernstein, Mike Carey, Stefano Ceri, W. Bruce Croft, David DeWitt, Mike Franklin, Hector Garcia Molina, Dieter Gawlick, Jim Gray, Laura Haas, Alon Halevy, Joe Hellerstein, Yannis Ioannidis, Martin Kersten, Michael Pazzani, Mike Lesk, David Maier, Jeff Naughton, Hans Schek, Timos Sellis, Avi Silberschatz, Mike Stonebraker, Rick Snodgrass, Jeff Ullman, Gerhard Weikum, Jennifer Widom, and Stan Zdonik. The Lowell database research self-assessment. *Commun. ACM*, 48(5):111–118, 2005.

[ABH09] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. Column-oriented database systems, 2009.

[ACD02] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. DBXplorer: A System for Keyword-Based Search over Relational Databases. *Data Engineering, International Conference on*, 0:0005, 2002.

[AM05a] V. N. Anh and A. Moffat. Simplified Similarity Scoring Using Term Ranks. In *Proceedings of the International Conference on Information Retrieval (ACM SIGIR)*, pages 226–233, Salvador, Brazil, 2005.

[AM05b] Vo Ngoc Anh and Alistair Moffat. Inverted Index Compression Using Word-Aligned Binary Codes. *Inf. Retr.*, 8(1):151–166, 2005.

[AM05c] Vo Ngoc Anh and Alistair Moffat. Inverted Index Compression Using Word-Aligned BinaryCodes. *Information Retrieval*, 8(1):151–166, 2005.

[AYBC$^+$06] Sihem Amer-Yahia, Chavdar Botev, Stephen Buxtonand Pat Case, Jochen Doerre, Mary Holstege, Darin McBeath, Michael Rys, and Jayavel Shanmugasundaram. XQuery 1.0 and XPath 2.0 Full-Text. Technical report, World Wide Web Consortium, May 2006.

[AYCR$^+$05] S. Amer-Yahia, P. Case, T. Rölleke, J. Shanmugasundaram, and G. Weikum. Report on the DB/IR Panel at Sigmod 2005. *SIGMOD Record*, 34(4):71–74, 2005.

[AYHR$^+$08] Sihem Amer-Yahia, Djoerd Hiemstra, Thomas Roelleke, Divesh Srivastava, and Gerhard Weikum. DB&IR integration: report on the Dagstuhl seminar "ranked XML querying". *SIGMOD Record*, 37(3):46–49, 2008.

[AyS05] Sihem Amer-yahia and Jayavel Shanmugasundaram. XML Full-Text Search: Challenges and Opportunities. In *Proceedings of 31th International Conference on Very Large Data Bases*, 2005.

[Bau99] Peter Baumann. A Database Array Algebra for Spatio-Temporal Data and Beyond. In *Next Generation Information Technologies and Systems*, pages 76–93, 1999.

[BB07] Roi Blanco and Alvaro Barreiro. Boosting static pruning of inverted files. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 777–778, New York, NY, USA, 2007. ACM.

[BCCM93] Eric W. Brown, James P. Callan, W. B Croft, and J. E.B. Moss. Supporting Full-Text Information Retrieval with a Persistent Object Store. Technical report, Amherst, MA, USA, 1993.

[BCG02] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Trans. Database Syst.*, 27(2):153–187, 2002.

[BCH$^+$03] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the Conference of Information and Knowledge Management (CIKM)*, pages 426–434, New Orleans, LA, USA, 2003.

[BDF$^+$98] Peter Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system RasDaMan. *SIGMOD Rec.*, 27(2):575–577, 1998.

[BDJ99] Michael W. Berry, Zlatko Drmac, and Elizabeth R. Jessup. Matrices, Vector Spaces, and Information Retrieval. *SIAM Review*, 41(2):335–362, 1999.

[BdVB99] H.E. Blok, Arjen P. de Vries, and H.M. Blanken. Top N MM query optimization: The best of both IR and DB worlds. In P. de Bra and L. Hardman, editors, *Conferentie Informatiewetenschap 1999 [Eng.: Information Science Conference 1999]*, Amsterdam, The Netherlands, December 1999. Werkgemeenschap Informatiewetenschap.

[BF95] Michael W. Berry and Ricardo D. Fierro. Low-Rank Orthogonal Decompositions for Information Retrieval Applications. Technical report, Knoxville, TN, USA, 1995.

[BGMP92] D. Barbara, H. Garcia-Molina, and D. Porter. The Management of Probabilistic Data. *IEEE Transactions on Knowledge and Data Engineering*, 4:487–502, 1992.

[BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[BGvK$^+$06] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery. In *Proc. SIGMOD*, pages 479–490, 2006.

[BK95] Peter Boncz and Martin Kersten. Monet: An Impressionist Sketch of an Advanced Database System. In *Proceedings Basque International Workshop on Information Technology*, San Sebastian, Spain, July 1995.

[BK99] Peter Boncz and Martin Kersten. MIL Primitives for Querying a Fragmented World. *The VLDB Journal*, 8(2):101–119, October 1999. The original publication is available in LINK, Springer-Verlag.

[BK03] J. Becker and D. Kuropka. Topic-based Vector Space Model. In *Proceedings of the 6th International Conference on Business Information Systems*, pages 7–12, Colorado Springs, July 2003.

[BL85] Chris Buckley and Alan F. Lewit. Optimization of inverted vector searches. In *SIGIR '85: Proceedings of the 8th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 97–110, New York, NY, USA, 1985. ACM.

[Bla88] David C. Blair. An extended relational document retrieval model. *Inf. Process. Manage.*, 24(3):349–371, 1988.

[bla02] An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Softw.*, 28(2):135–151, 2002.

[BLS$^+$94] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension Syntax. *SIGMOD Record*, 23(1):87–96, 1994.

[BMR$^+$06] Henk Ernst Blok, Vojkan Mihajlović, Georgina Ramírez, Thijs Westerveld, Djoerd Hiemstra, and Arjen P. de Vries. The tijah xml information retrieval system. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '06, pages 725–725, New York, NY, USA, 2006. ACM.

[BNJ03] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, 2003.

[Bon02] Peter Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications.* Ph.d. thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, May 2002.

[BZN05] Peter Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proceedings of the Conference of Innovative Database Research (CIDR)*, pages 225–237, Asilomar, CA, USA, 2005.

[CCG00] P. Ciaccia, Roberto Cornacchia, and A. Ghidini. Optimization and Evaluation of Generalized Top Queries. In *Proceeding of the Italian Symposium on Advanced Database Systems (SEBD)*, pages 273–287, L'Aquila, Italy, 2000.

[CCH92] James P. Callan, W. Bruce Croft, and Stephen M. Harding. The INQUERY Retrieval System. In *Proc. DEXA*, pages 78–83, 1992.

[CDHW06] Surajit Chaudhuri, Gautam Das, Vagelis Hristidis, and Gerhard Weikum. Probabilistic information retrieval approach for ranking of database query results. *ACM Trans. Database Syst.*, 31(3):1134–1168, 2006.

[CdV06] Roberto Cornacchia and Arjen P. de Vries. A declarative DB-powered approach to IR. In *Proceedings of the European Conference on IR Research (ECIR)*, London, United Kingdom, April 2006.

[CdV07] Roberto Cornacchia and Arjen P. de Vries. A Parameterised Search System. In *Proceedings of the European Conference on IR Research (ECIR)*, Rome, Italy, April 2007.

[CdVR08] Maarten Clements, Arjen P. de Vries, and Marcel J.T. Reinders. Optimizing single term queries using a personalized Markov random walk over the social graph. In *ECIR Workshop on Exploiting Semantic Annotations in Information Retrieval(ESAIR'08)*, 2008.

[CDY95] Surajit Chaudhuri, Umeshwar Dayal, and Tak W. Yan. Join queries with external text sources: execution and optimization techniques. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 410–422, New York, NY, USA, 1995. ACM.

[CGM90] Upen S. Chakravarthy, John Grant, and Jack Minker. Logic-based approach to semantic query optimization. *ACM Trans. Database Syst.*, 15(2):162–207, 1990.

[Cha98] Surajit Chaudhuri. An overview of query optimization in relational systems. In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43, New York, NY, USA, 1998. ACM.

[CHZ$^+$08] Roberto Cornacchia, S. Héman, M. Zukowski, Arjen P. de Vries, and Peter Boncz. Flexible and efficient IR using Array Databases. *VLDB Journal, special issue on IR&DB integration*, 17(1):151–168, 2008.

[CK85] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. In *SIGMOD '85: Proceedings of the 1985 ACM SIGMOD international conference on Management of data*, pages 268–279, New York, NY, USA, 1985. ACM.

[CK97] Michael J. Carey and Donald Kossmann. On saying "Enough already!" in SQL. *SIGMOD Rec.*, 26(2):219–230, 1997.

[CK98] Michael J. Carey and Donald Kossmann. Reducing the Braking Distance of an SQL Query Engine. In *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 158–169, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[CMKL$^+$09] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla,

D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker, and S. Zdonik. A demonstration of SciDB: a science-oriented DBMS. *Proc. VLDB Endow.*, 2(2):1534–1537, 2009.

[CMWM09] Philippe Cudré-Mauroux, Eugene Wu, and Samuel Madden. The Case for RodentStore: An Adaptive, Declarative Storage System. In *CIDR 2009: Fourth Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, USA, 2009. www.cidrdb.org.

[Cod79] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst.*, 4(4):397–434, 1979.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[Cor06] Roberto Cornacchia. Querying Sparse Matrices for Information Retrieval. In *Proceedings of the British National Conference on Databases (BNCOD)*, 2006. PhD Forum.

[CP85] S. Ceri and G. Pelagatti. *Distributed Databases.* McGraw-Hill Book Company, Singapore, 1985.

[CP87] Roger Cavallo and Michael Pittarelli. The Theory of Probabilistic Databases. In *VLDB '87: Proceedings of the 13th International Conference on Very Large Data Bases*, pages 71–81, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.

[Cra81] Robert G. Crawford. The relational model in information retrieval. *Journal of the American Society for Information Science*, 32(1):51–64, 1981.

[CRW05] S. Chaudhuri, R. Ramakrishnan, and G. Weikum. Integrating DB and IR Technologies: What is the Sound of One Hand Clapping? In *Proc. CIDR*, pages 1–12, Asilomar, CA, USA, 2005.

[CSS05] C. L. A. Clarke, F. Scholer, and I. Soboroff. The TREC 2005 Terabyte Track. In *Proceedings of the Text Retrieval Conference (TREC)*, Gaithersburg, MD, USA, 2005.

[CST92] W. Bruce Croft, Lisa A. Smith, and Howard R. Turtle. A loosely-coupled integration of a text retrieval system and an object-oriented database system. In *SIGIR '92: Proceedings of the 15th* *annual international ACM SIGIR conference on Research and development in information retrieval*, pages 223–232, New York, NY, USA, 1992. ACM.

[CvBdV04] Roberto Cornacchia, Alex van Ballegooij, and Arjen P. de Vries. A Case Study on Array Query Optimisation. In *Proceedings of the First International Workshop on Computer Vision meets Databases (CVDB)*, pages 3–10, Paris, France, June 2004. ACM Press. In cooperation with ACM SIGMOD.

[CWI] CWI Amsterdam and University of Amsterdam. MonetDB. `http://monetdb.cwi.nl`.

[CWLL09] Yi Chen, Wei Wang, Ziyang Liu, and Xuemin Lin. Keyword search on structured and semi-structured data. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 1005–1010, New York, NY, USA, 2009. ACM.

[DDL$^+$90] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by Latent Semantic Analysis. *JASIS*, 41(6):391–407, 1990.

[DDRvdV00] James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst. *Templates for the solution of algebraic eigenvalue problems: a practical guide.* SIAM, 2000.

[DER86] Iain S Duff, Albert M Erisman, and John K Reid. *Direct methods for sparse matrices.* Oxford University Press, Inc., New York, NY, USA, 1986.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *OSDI*, 2004.

[DL89] P. Dadam and V. Linnemann. Advanced information management (AIM): advanced database technology for integrated applications. *IBM Syst. J.*, 28(4):661–681, 1989.

[DS04] Nilesh Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 864–875. VLDB Endowment, 2004.

[dV00] Arjen P. de Vries. Challenging Ubiquitous Inverted Files. In *Proceedings of the 1st DELOS Network of Excellence Workshop on Information Seeking, Searching and Querying in Digital Libraries, Zurich, Switzerland*, volume 01/W001 of *ERCIM Workshop Proceedings*, page 13, Biot, France, December 2000. European Research Consortium for Informatics and Mathematics (ERCIM).

[dV01] Arjen P. de Vries. Content Independence in Multimedia Databases. *JASIST*, 52(11):954–960, September 2001.

[dVMNK01] Arjen P. de Vries, N. Mamoulis, N.J. Nes, and Martin Kersten. Efficient image retrieval by exploiting vertical fragmentation. Technical Report INS-R0109, CWI, November 2001.

[dVMNK02] Arjen P. de Vries, Nikos Mamoulis, Niels Nes, and Martin Kersten. Efficient k-NN search on vertically decomposed data. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 322–333, New York, NY, USA, 2002. ACM.

[EMK$^+$04] Andrew Eisenberg, Jim Melton, Krishna Kulkarni, Jan-Eike Michels, and Fred Zemke. SQL:2003 has been published. *SIGMOD Rec.*, 33(1):119–126, 2004.

[Ext] Extractiv. Turn web content into structured data. `http://www.extractiv.com`.

[Fac] Facebook. Facebook. `http://www.facebook.com`.

[Fag99] Ronald Fagin. Combining fuzzy information from multiple systems. *J. Comput. Syst. Sci.*, 58(1):83–99, 1999.

[FB99] P. Furtado and Peter Baumann. Storage of Multidimensional Arrays based on Arbitrary Tiling. In *Proc. of the 15th International Conference on Data Engineering, ICDE99*, pages 408–489, March 1999.

[FDD$^+$88] George W. Furnas, Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, Richard A. Harshman, Lynn A. Streeter, and Karen E. Lochbaum. Information Retrieval using a Singular Value Decomposition Model of Latent Semantic Structure. In Yves Chiaramella, editor, *SIGIR*, pages 465–480. ACM, 1988.

[FG99] Paolo Ferragina and Roberto Grossi. The string B-tree: a new data structure for string search in external memory and its applications. *J. ACM*, 46(2):236–280, 1999.

[FKM00] Daniela Florescu, Donald Kossmann, and Ioana Manolescu. Integrating keyword search into XML query processing. *Computer Networks*, 33(1-6):119–135, 2000.

[FLN01] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *PODS '01: Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 102–113, New York, NY, USA, 2001. ACM.

[FR97] Norbert Fuhr and Thomas Rölleke. A Probabilistic Relational Algebra for the Integration of Information Retrieval and Database Systems. *ACM Trans. Inf. Syst.*, 15(1):32–66, 1997.

[Fuh90] Norbert Fuhr. A Probabilistic Framework for Vague Queries and Imprecise Information in Databases. In *VLDB '90: Proceedings of the 16th International Conference on Very Large Data Bases*, pages 696–707, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[Fuh96a] Norbert Fuhr. Models for Integrated Information Retrieval and Database Systems. *IEEE Data Eng. Bull.*, 19(1):3–13, 1996.

[Fuh96b] Norbert Fuhr. Object-Oriented and Database Concepts for the Design of Networked Information Retrieval Systems. In *Proc. CIKM*, pages 164–172, 1996.

[Fuh00] Norbert Fuhr. Probabilistic Datalog: implementing logical information retrieval for advanced applications. *J. Am. Soc. Inf. Sci.*, 51(2):95–110, 2000.

[Gal] Galago. Galago. `http://www.galagosearch.org`.

[GB07] Angelica Garcia Gutierrez and Peter Baumann. Modeling Fundamental Geo-Raster Operations with Array Algebra. In *ICDMW '07: Proceedings of the Seventh IEEE International Conference on Data Mining Workshops*, pages 607–612, Washington, DC, USA, 2007. IEEE Computer Society.

[GBK00] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Optimizing Multi-Feature Queries for Image Databases. In , pages 419–428, 2000.

[GBK01] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Towards Efficient Multi-Feature Queries in Heterogeneous Environments. *Information Technology: Coding and Computing, International Conference on*, 0:0622, 2001.

[GBS04] Torsten Grabs, Klemens Bhoem, and Hans-Jorg Schek. PowerDB-IR: Scalable Information Retrieval and Storage with a Cluster of Databases. *Knowledge and Information Systems*, 6(4):465–505, 2004.

[GFHR97] D. A. Grossman, O. Frieder, D. O. Holmes, and D. C. Roberts. Integrating Structured Data and Text: A Relational Approach. *JASIS*, 48(2):122–132, 1997.

[GL83] G. H. Golub and C. F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, Maryland, 1983.

[GLR97] César A. Galindo-Legaria and Arnon Rosenthal. Outerjoin simplification and reordering for query optimization. *ACM Trans. Database Syst.*, 22(1):43–74, 1997.

[Goo] Google. Google Web Search. `http://www.google.com`.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, 1993.

[Gra94] Goetz Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[GSBS03] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. XRANK: ranked keyword search over XML documents. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 16–27, New York, NY, USA, 2003. ACM.

[GvKT03] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase join: teach a relational DBMS to watch its (axis) steps. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 524–535. VLDB Endowment, 2003.

[Haw04] David Hawking. Challenges in Enterprise Search. In *Proc. ADC*, pages 15–26, 2004.

[HBYFL92] Donna Harman, R. Baeza-Yates, Edward Fox, and W. Lee. *Inverted files*, volume Information retrieval: data structures and algorithms, pages 28–43. Prentice-Hall, Inc., 1992.

[HG04] Erik Hatcher and Otis Gospodnetic. *Lucene in Action.* Manning Publications Co., Greenwich, CT, USA, 2004.

[HGP03] Vagelis Hristidis, Luis Gravano, and Yannis Papakonstantinou. Efficient IR-style keyword search over relational databases. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 850–861. VLDB Endowment, 2003.

[Hie98] D. Hiemstra. A Linguistically Motivated Probabilistic Model of Information Retrieval. In *ECDL*, pages 569–584, 1998.

[Hil91] David Hilbert. Ueber die stetige Abbildung einer Line auf ein Flchenstck. *Mathematische Annalen*, 38:459–460, 1891. 10.1007/BF01199431.

[HM04] B. Howe and D. Maier. Algebraic Manipulation of Scientific Datasets. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 924–935, Toronto, Canada, 2004.

[HN02] Arvind Hulgeri and Charuta Nakhe. Keyword Searching and Browsing in Databases using BANKS. In *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*, page 431, Washington, DC, USA, 2002. IEEE Computer Society.

[How07] Bill Howe. *Gridfields: model-driven data transformation in the physical sciences.* PhD thesis, Portland, OR, USA, 2007.

[HP02] Vagelis Hristidis and Yannis Papakonstantinou. Discover: keyword search in relational databases. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 670–681. VLDB Endowment, 2002.

[HRS07] Harry Halpin, Valentin Robu, and Hana Shepherd. The complex dynamics of collaborative tagging. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 211–220, New York, NY, USA, 2007. ACM.

[HRvOF06] D. Hiemstra, H. Rode, R. van Os, and J. Flokstra. PFTijah: text search in an XML database system. In *Proc. OSIR*, 2006.

[IK84] Toshihide Ibaraki and Tiko Kameda. On the optimal nesting order for computing N-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, 1984.

[IKM09] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Self-organizing tuple reconstruction in column-stores. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 297–308, New York, NY, USA, 2009. ACM.

[IL84] Tomasz Imieliński and Jr. Witold Lipski. Incomplete Information in Relational Databases. *J. ACM*, 31(4):761–791, 1984.

[JK02] Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of IR techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446, 2002.

[JO95] Byeong-Soo Jeong and Edward Omiecinski. Inverted File Partitioning Schemes in Multiple Disk Systems. *IEEE Trans. Parallel Distrib. Syst.*, 6(2):142–153, 1995.

[Joh90] David Johnson. Local optimization and the Traveling Salesman Problem. In Michael Paterson, editor, *Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 446–461. Springer Berlin / Heidelberg, 1990. 10.1007/BFb0032050.

[KIML11] M. L. Kersten, S. Idreos, S. Manegold, and E. Liarou. The Researchers Guide To The Data Deluge: Querying A Scientific Database In Just A Few Seconds. In *Proceedings of International Conference on Very Large Data Bases 2011 (VLDB)*, pages 585 – 597, 2011.

[KKNR04] Raghav Kaushik, Rajasekar Krishnamurthy, Jeffrey F. Naughton, and Raghu Ramakrishnan. On the integration of structure indexes and inverted lists. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 779–790, New York, NY, USA, 2004. ACM.

[Knu97] Donald E. Knuth. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[KPS97] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. A Relational Approach to the Compilation of Sparse Matrix Programs. In *Euro-Par*, pages 318–327, 1997.

[LD10] Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2010.

[LFW09] Guoliang Li, Jianhua Feng, and Jianyong Wang. Structure-aware indexing for keyword search in databases. In *CIKM '09: Proceeding of the 18th ACM conference on Information and knowledge management*, pages 1453–1456, New York, NY, USA, 2009. ACM.

[LHKK79] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.*, 5(3):308–323, 1979.

[Lib] LibraryThing. LibraryThing. `http://www.librarything.com`.

[Lin] LinkedIn Corp. LinkedIn. `http://www.linkedin.com`.

[Lip79] Jr. Witold Lipski. On semantic issues connected with incomplete information databases. *ACM Trans. Database Syst.*, 4(3):262–296, 1979.

[LLWZ07] Yi Luo, Xuemin Lin, Wei Wang, and Xiaofang Zhou. Spark: top-k keyword query in relational databases. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 115–126, New York, NY, USA, 2007. ACM.

[LMR$^+$05] J. List, V. Mihajlovic, G. Ramirez, Arjen P. de Vries, D. Hiemstra, and H.E. Blok. TIJAH: Embracing IR Methods in XML Database. *Information Retrieval*, 8(4):547–570, December 2005.

[LMW96] L. Libkin, R. Machlin, and L. Wong. A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques. In *ACM SIGMOD 1996*, pages 228–239. ACM Press, June 1996.

[LS03] Alberto Lerner and Dennis Shasha. AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 345–356, Berlin, Germany, 2003.

[LYMC06] Fang Liu, Clement Yu, Weiyi Meng, and Abdur Chowdhury. Effective keyword search in relational databases. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 563–574, New York, NY, USA, 2006. ACM.

[Mac79] Ian A. Macleod. SEQUEL as a Language for Document Retrieval. *Journal of the American Society for Information Science*, 30(5):243–249, 1979.

[Mac07] Rona Machlin. Index-based multidimensional array queries: safety and equivalence. In *PODS '07: Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 175–184, New York, NY, USA, 2007. ACM.

[Mat] MathWorks Inc. Matlab. `http://www.mathworks.com`.

[Mel08] Massimo Melucci. A basis for information retrieval in context. *ACM Trans. Inf. Syst.*, 26(3):1–41, 2008.

[MGC07] Mauricio Marin and Veronica Gil-Costa. High-performance distributed inverted files. In *CIKM '07: Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 935–938, New York, NY, USA, 2007. ACM.

[Mih06] V. Mihajlovic. *Score Region Algebra. A Flexible Framework for Structured Information Retrieval.* PhD thesis, University of Twente, 2006.

[MMR00] A. MacFarlane, J. A. McCann, and S. E. Robertson. Parallel Search using Partitioned Inverted Files. In *SPIRE '00: Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00)*, page 209, Washington, DC, USA, 2000. IEEE Computer Society.

[Mor66] G.M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, Ottawa, Ontario, 1966.

[MR96] Rajeev Motwani and Prabhakar Raghavan. Randomized algorithms. *ACM Comput. Surv.*, 28(1):33–37, 1996.

[MRS08] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval.* Cambridge University Press, 2008.

[MS97] A.P. Marathe and K. Salem. A Language For Manipulating Arrays. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 46–55, Athens, Greece, 1997.

[MV93] David Maier and Bennet Vance. A call to order. In *PODS '93: Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 1–16, New York, NY, USA, 1993. ACM.

[MWZ06] Alistair Moffat, William Webber, and Justin Zobel. Load balancing for term-distributed parallel retrieval. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 348–355, New York, NY, USA, 2006. ACM.

[MYP09] Alexander Markowetz, Yin Yang, and Dimitris Papadias. Keyword search over relational tables and streams. *ACM Trans. Database Syst.*, 34(3):1–51, 2009.

[MyS] MySpace. MySpace.com. `http://www.myspace.com`.

[MZ94] Alistair Moffat and Justin Zobel. Self-Indexing Inverted Files. In *Australasian Database Conference*, pages 79–91, 1994.

[Ng01] Wilfred Ng. An extension of the relational data model to incorporate ordered domains. *ACM Trans. Database Syst.*, 26(3):344–383, 2001.

[NR99] Surya Nepal and M.V. Ramakrishna. Query Processing Issues in Image(Multimedia) Databases. *Data Engineering, International Conference on*, 0:22, 1999.

[OAP$^+$06] I. Ounis, G. Amati, V. Plachouras, B. He, C. Macdonald, and C. Lioma. Terrier: A High Performance and Scalable Information Retrieval Platform. In *Proc. OSIR*, 2006.

[OC01] Paul Ogilvie and James P. Callan. Experiments Using the Lemur Toolkit. In *TREC*, 2001.

[Oxf] Oxford University Press. Oxford English Dictionary online. `http://www.oed.com`.

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[PC98] J. Ponte and W. Croft. A language modelling approach to information retrieval. In *Proceedings of the 21st ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '98)*, 1998.

[PD89] Peter Pistor and Peter Dadam. The Advanced Information Management Prototype. In *Papers from the Workshop "Theory and Applications of Nested Relations and Complex Objects' on Nested Relations and Complex Objects*, pages 3–26, London, UK, 1989. Springer-Verlag.

[Pea90] G. Peano. Sur une courbe, qui remplit toute une aire plane. *MA*, 36:157–160, 1890.

[PFLvR10] Benjamin Piwowarski, Ingo Frommholz, Mounia Lalmas, and Keith van Rijsbergen. What can Quantum Theory bring to ÏR̈? In Jimmy Huang, Nick Koudas, Gareth Jones, Xindong Wu, Kevyn Collins-Thompson, and Aijun An, editors, *CIKM'10: Proceedings of the nineteenth ACM conference on Conference on information and knowledge management.* ACM, 2010.

[PGL02] Martin Pelikan, David E. Goldberg, and Fernando G. Lobo. A Survey of Optimization by Building and Using Probabilistic Models. *Computational Optimization and Applications*, 21:5–20, 2002. 10.1023/A:1013500812258.

[PIPAFF09] Joaquín Pérez-Iglesias, José R. Pérez-Agüera, Víctor Fresno, and Yuval Z. Feinstein. Integrating the Probabilistic Models BM25/BM25F into Lucene. *CoRR*, abs/0911.5046, 2009.

[Por80] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[Put91] Steve Putz. Using a Relational Database for an Inverted Text Index, 1991.

[QYC09] Lu Qin, Jeffrey Xu Yu, and Lijun Chang. Keyword search in databases: the power of RDBMS. In *SIGMOD '09: Proceedings of the*

35th SIGMOD international conference on Management of data, pages 681–694, New York, NY, USA, 2009. ACM.

[R] R. The R Project for Statistical Computing. `http://www.r-project.org/`.

[RSJ88] Stephen E. Robertson and Karen Sparck Jones. *Relevance weighting of search terms*, pages 143–160. Taylor Graham Publishing, London, UK, UK, 1988.

[RT04] R.A.O'Keefe and A. Trotman. The simplest query language that could possibly work. In *Proc. INEX*, 2004.

[RTK05] T. Rölleke, T. Tsikrika, and G. Kazai. A General Matrix Framework for Modelling Information Retrieval. *IP&M*, 42(1):4–30, 2005.

[Run88] Arnon Rungsawang. DSIR: the First TREC-7 Attempt, 1988.

[RWB98] S. E. Robertson, S. Walker, and M. Beaulieu. Okapi at TREC-7: automatic ad hoc, filtering, VLC and interactive track. In *Proceedings of the Text Retrieval Conference (TREC)*, pages 143–167, Gaithersburg, MD, USA, 1998.

[RWWA08] Thomas Rölleke, Hengzhi Wu, Jun Wang, and Hany Azzam. Modelling retrieval models in a probabilistic relational algebra with a new operator: the relational Bayes. *The VLDB Journal*, 17(1):5–37, 2008.

[RZ09] Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: Bm25 and beyond. *Found. Trends Inf. Retr.*, 3:333–389, April 2009.

[SAB^{+}05] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: a column-oriented DBMS. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[SAH87] Michael Stonebraker, Jeff Anton, and Eric Hanson. Extending a database system with procedures. *ACM Trans. Database Syst.*, 12(3):350–376, 1987.

[Sal71] G. Salton. *The SMART Retrieval System—Experiments in Automatic Document Processing.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1971.

[Sci] SciLens. SciLens, Database technology to advance science. `http://www.scilens.org`.

[SM86] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, Inc., New York, NY, USA, 1986.

[SM97] Wolfgang Scheufele and Guido Moerkotte. On the complexity of generating optimal plans with cross products (extended abstract). In *PODS '97: Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 238–248, New York, NY, USA, 1997. ACM.

[SMK93] M. Steinbrunn, G. Moerkotte, and A. Kemper. Optimizing Join Orders. Technical report, University of Passau, 1993.

[SMTC04] T. Strohman, D. Metzler, H. Turtle, and W.B. Croft. Indri: A language model-based search engine for complex queries. In *Proceedings of the Int. Conf. on Intelligence Analysis*, 2004.

[SP82] Hans-Jörg Schek and Peter Pistor. Data Structures for an Integrated Data Base Management and Information Retrieval System. In *VLDB '82: Proceedings of the 8th International Conference on Very Large Data Bases*, pages 197–207, San Francisco, CA, USA, 1982. Morgan Kaufmann Publishers Inc.

[Sph] Sphinx. Sphinx SQL full-text search engine. `http://www.sphinxsearch.com`.

[SS92] Banger Skillicorn and D. B. Skillicorn. Flat Arrays as a Categorical Data Type. Technical report, 1992.

[Ste09] William Stein. Public-key Cryptography. In *Elementary Number Theory: Primes, Congruences, and Secrets*, Undergraduate Texts in Mathematics, pages 1–20. Springer New York, 2009.

[Str08] Trevor Strohman. *Efficient processing of complex features for information retrieval.* PhD thesis, 2008. AAI3315499.

[SWY75] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, 1975.

[TBM$^+$08] Martin Theobald, Holger Bast, Debapriyo Majumdar, Ralf Schenkel, and Gerhard Weikum. TopX: efficient and versatile top-k query processing for semistructured data. *The VLDB Journal*, 17(1):81–115, 2008.

[TE08] Kamal Taha and Ramez Elmasri. CXLEngine: a comprehensive XML loosely structured search engine. In *DataX '08: Proceedings of the 2008 EDBT workshop on Database technologies for handling XML information on the web*, pages 37–42, New York, NY, USA, 2008. ACM.

[TE09] Kamal Taha and Ramez Elmasri. XCDSearch: An XML Context-Driven Search Engine. *IEEE Transactions on Knowledge and Data Engineering*, 99(PrePrints), 2009.

[TJ95] H. Turtle and J.Flood. Query evaluation: strategies and optimizations. *Information Processing and Management*, 31(6):831–850, 1995.

[Tro03] Andrew Trotman. Compressing Inverted Files. *Inf. Retr.*, 6(1):5–19, 2003.

[TSB09] Nan Tang, Lefteris Sidirourgos, and Peter Boncz. Space-economical partial gram indices for exact substring matching. In *CIKM '09: Proceeding of the 18th ACM conference on Information and knowledge management*, pages 285–294, New York, NY, USA, 2009. ACM.

[TSW05] Martin Theobald, Ralf Schenkel, and Gerhard Weikum. An efficient and versatile query engine for topx search. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 625–636. VLDB Endowment, 2005.

[Tur91] Howard Robert Turtle. *Inference networks for document retrieval.* PhD thesis, Amherst, MA, USA, 1991.

[TWS04] Martin Theobald, Gerhard Weikum, and Ralf Schenkel. Top-k query evaluation with probabilistic guarantees. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 648–659. VLDB Endowment, 2004.

[Vas79] Yannis Vassiliou. Null values in data base management a denotational semantics approach. In *SIGMOD '79: Proceedings of the*

1979 ACM SIGMOD international conference on Management of data, pages 162–169, New York, NY, USA, 1979. ACM.

[Vas98] Panos Vassiliadis. Modeling Multidimensional Databases, Cubes and Cube Operations. In *SSDBM '98: Proceedings of the 10th International Conference on Scientific and Statistical Database Management*, pages 53–62, Washington, DC, USA, 1998. IEEE Computer Society.

[vB09] Alex van Ballegooij. *RAM: Array Database Management through Relational Mapping.* PhD thesis, Universiteit van Amsterdam, 9 2009. SIKS 2009-25.

[vBCdV05] Alex van Ballegooij, Roberto Cornacchia, and Arjen P. de Vries. Automatic optimization of array queries. Technical Report INS-E0507, CWI, Amsterdam, The Netherlands, April 2005.

[vBCdVK05] Alex van Ballegooij, Roberto Cornacchia, Arjen P. de Vries, and Martin Kersten. Distribution Rules for Array Database Queries. In *Proceedings of the International Conference on Database and Expert*, volume 3588, pages 55–64. Springer-Verlag GmbH, August 2005.

[Vec] VectorWise. Ingres/VectorWise. `http://www.vectorwise.com`.

[vR79] C. J. van Rijsbergen. *Information Retrieval, 2nd edition.* Butterworths, London, 1979.

[vR04] C. J. van Rijsbergen. *The Geometry of Information Retrieval.* Cambridge University Press, New York, NY, USA, 2004.

[VS99] Panos Vassiliadis and Timos Sellis. A survey of logical models for OLAP databases. *SIGMOD Rec.*, 28(4):64–69, 1999.

[WC06] Xing Wei and W. Bruce Croft. LDA-based document models for ad-hoc retrieval. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 178–185, New York, NY, USA, 2006. ACM.

[WCY$^+$10] Jun Wang, Maarten Clements, Jie Yang, Arjen P. de Vries, and Marcel J. T. Reinders. Personalization of tagging systems. *Inf. Process. Manage.*, 46(1):58–70, 2010.

[Wei07] Gerhard Weikum. DB&IR: both sides now. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 25–30, New York, NY, USA, 2007. ACM.

[Wik] Wikipedia. 2009 flu pandemic. `http://en.wikipedia.org/wiki/2009_flu_pandemic`.

[WLMZ03] Ji-Rong Wen, Qing Li, Wei-Ying Ma, and HongJiang Zhang. A Multi-paradigm Querying Approach for a Generic Multimedia Database Management System. *SIGMOD Record*, 32(1):26–34, 2003.

[WPZ$^+$06] Shan Wang, Zhaohui Peng, Jun Zhang, Lu Qin, Sheng Wang, Jeffrey Xu Yu, and Bolin Ding. NUITS: a novel user interface for efficient keyword search over databases. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 1143–1146. VLDB Endowment, 2006.

[WZW85] S. K. M. Wong, Wojciech Ziarko, and Patrick C. N. Wong. Generalized vector spaces model in information retrieval. In *SIGIR '85: Proceedings of the 8th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 18–25, New York, NY, USA, 1985. ACM.

[YDS09] Hao Yan, Shuai Ding, and Torsten Suel. Compressing term positions in web indexes. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '09, pages 147–154, New York, NY, USA, 2009. ACM.

[You] YouTube, LLC. YouTube. `http://www.youtube.com`.

[ZBNH05] M. Zukowski, Peter Boncz, N. Nes, and S. Héman. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Engineering Bulletin*, 28(2):17–22, June 2005.

[ZHNB06] M. Zukowski, S. Héman, N. Nes, and Peter Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the International Conference of Data Engineering (IEEE ICDE)*, Atlanta, GA, USA, April 2006.

[ZHY09] Yi Zhang, Herodotos Herodotou, and Jun Yang. RIOT: I/O-Efficient Numerical Computing without SQL. In *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*. www.crdrdb.org, 2009.

[ZM98] Justin Zobel and Alistair Moffat. Exploring the Similarity Space. *SIGIR Forum*, 32(1):18–34, 1998.

[ZM06] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006.

[ZMR98] Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Trans. Database Syst.*, 23(4):453–490, 1998.

[ZND+01] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On supporting containment queries in relational database management systems. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 425–436, New York, NY, USA, 2001. ACM.

Summary

Enter some keywords and a magic box will find documents for you. Only relatively simple variations of this familiar search scenario for digital systems have reached the large audience in the past two decades. Search technology has however become more complex over the years, with new and improved theoretical models and algorithms hidden behind the “Search” button. For example, new models can better exploit word correlations or thesauri to improve search quality, while algorithms and data structures can use parallelisation and compression techniques. Search has evolved in other directions, too. More and more contextual information is exploited during the search process. Online stores can refine product retrieval based on user profiles, behaviour history and similarity with other users. Also, search has extended beyond the scope of “simple” text search. The diversity of evidence types used by modern systems includes text, markup languages, emails, audio/video material, online social content such as blogs or social network posts, to mention a few.

We view each specific setup of the search system as a search application. The development of such specialised applications requires considerable effort even to highly skilled information retrieval engineers, who need to apply the latest laboratory results to a growing variety of real-world complex scenarios [Haw04]. This thesis investigates how the architecture design of a search system could be reconsidered to simplify the creation of these custom search applications, focusing on two main concepts: *content independence* and *data independence*. Content independence [dVMNK01, Mih06, WLMZ03] is referred to as the decoupling between search methods and logical content representation (e.g. ranking XML data should not be different from ranking flat text data). Data independence is referred to as the decoupling between algorithms and data organisation. Absence of these two properties yields search systems that are inflexible with respect to changes in the expected range of tasks, contextual information, content representation and

physical data organisation.

This thesis proposes an innovation in the search system engineering process, by introducing a layered approach typical of database systems, that allows to enable both content and data independence. Content independence is achieved by expressing data and operations uniformly within the Matrix Framework for IR [RTK05]. Data independence is achieved by mapping this mathematical framework to database technology for actual data access: the SRAM abstraction layer allows IR engineers to "see matrices while touching relational tables". Information retrieval tasks can be specified using a matrix-based query language, in a way that directly corresponds to the underlying mathematical formulas. SRAM translates and optimises array queries into equivalent relational queries that can be executed on relational database systems.

A second goal of this research is to investigate how such an increased flexibility can be obtained without significant performance losses. Previous attempts of using database technology for supporting information retrieval tasks have shown repeatedly that the runtime efficiency of such solutions could not compete with specialised applications based on inverted file structures [CRW05]. The results presented in the evaluation section of this thesis, which show top-level performance and scalability scores, are particularly significant in that the database queries used were obtained automatically from a declarative specification of the mathematical problem at hand. This research identifies two main requirements for achieving competitive efficiency in the software architecture proposed: *(i)* mapping from the array to the relational domains is optimised with a specific set of *array-aware transformation rules*, with support for *sparse arrays*; *(ii)* the underlying database engine is tuned for *modern hardware exploitation*, like the MonetDB/X100 database system [ZBNH05] used for this research, which uses *vectorised in-cache* query execution to achieve good CPU utilisation [BZN05] and a *column-oriented storage manager* that provides transparent *light-weight data compression* to improve I/O-bandwidth utilisation [ZHNB06].

Samenvatting

Type een paar trefwoorden en een magische doos vindt geschikte documenten: een bekend scenario voor zoeken in moderne computersystemen. Slechts een klein aantal, relatief simpele variaties op dit thema bereikten de laatste twee decennia het publiek. Zoektechnologie is echter door de jaren heen complexer en krachtiger geworden, met nieuwe en verbeterde theoretische modellen en algoritmes verborgen achter de “Zoek” knop.

Die nieuwe modellen maken bijvoorbeeld beter gebruik van verbanden tussen woorden of van thesauri om de kwaliteit van de resultaten te verbeteren. Algoritmes en data structuren gebruiken parallellisme en compressie voor betere efficiëntie. Zoeken is ook in andere richtingen gegroeid. Meer en meer contextuele informatie wordt benut tijdens het zoekproces. Online winkels verfijnen het zoeken in producten op basis van gebruikersprofielen, gedrag en overeenkomsten met andere gebruikers. Daarnaast is zoeken meer geworden dan enkel het “eenvoudige” zoeken in tekst. Moderne systemen baseren zoekresultaten op een diversiteit aan bronnen, zoals de tekst zelf, annotaties in verschillende formaten (bv. HTML, XML), maar ook e-mail, audio/video materiaal, en berichten op online sociale netwerken.

Elk zoeksysteem wordt vaak gezien als een losstaande applicatie. De ontwikkeling van zulke gespecialiseerde applicaties kost een aanzienlijke hoeveelheid tijd, zelfs voor ervaren ontwikkelaars. Zij moeten de nieuwste onderzoeksresultaten op een groeiend aantal in de praktijk voorkomende complexe scenario’s toepassen [Haw04]. Dit proefschrift beoogt zoeksystemen anders te ontwerpen, om zo het bouwen van zoekapplicaties te vereenvoudigen. Het richt zich op twee hoofdaspecten: *onafhankelijkheid van inhoud* en *onafhankelijkheid van data*. Onafhankelijkheid van inhoud [dVMNK01, Mih06, WLMZ03] betekent het scheiden van zoekmethoden van de logische representatie van de inhoud (het rangschikken van XML data moet bijvoorbeeld niet anders zijn dan het rangschikken van platte tekst). Onafhankelijkheid van data staat voor het scheiden van algoritmes en fysieke data organisatie. Omdat zoeksystemen worden gebouwd zonder deze twee aspecten in aanmerking te nemen, zijn gebouwde zoeksystemen vaak inflexibel met betrekking tot latere wijzigingen in de verwachte omvang van taken, contextuele informatie, representatie van de inhoud en fysieke data organisatie.

Dit proefschrift introduceert daarom een innovatie in het ontwikkelproces van zoeksystemen. Door het overnemen van de gelaagde aanpak die kenmerkend is voor databasesystemen, wordt zowel onafhankelijkheid van inhoud als onafhankelijkheid van data haalbaar. Onafhankelijkheid van inhoud wordt bereikt door data en algoritmiek uniform uit te drukken in het Matrix Framework for IR, een theoretisch raamwerk voor information retrieval (IR) op basis van matrices [RTK05]. Onafhankelijkheid van data wordt vervolgens gerealiseerd door uitdrukkingen in dit raamwerk te vertalen operaties in database systemen, door middel van de abstractielaag SRAM. Zo kunnen IR ontwikkelaars zich uitdrukken in matrices en toch relationele tabellen manipuleren. Information retrieval taken worden beschreven in een matrix-gebaseerde querytaal die een op een overeenkomt met de onderliggende wiskundige formules. SRAM vertaalt en optimaliseert operaties op arrays in equivalente relationele operaties zodat deze uitgevoerd kunnen worden op relationele database systemen.

Een tweede doel van dit proefschrift onderzoekt hoe deze grotere flexibiliteit verwezenlijkt kan worden zonder significant verlies in prestaties. Eerdere pogingen om databasetechnologie te gebruiken voor het ondersteunen van information retrieval systemen konden herhaaldelijk qua efficiëntie niet wedijveren met gespecialiseerde applicaties gebouwd op geïnverteerde lijsten [CRW05]. De in de evaluatie gepresenteerde resultaten tonen aan dat competitieve prestaties en schaalbaarheid realiseerbaar is; een opmerkelijke prestatie omdat de operaties op de onderliggende database automatisch gegenereerd worden uit een declaratieve specificatie van het wiskundige probleem. Het onderzoek identificeert twee belangrijke vereisten voor het behalen van de benodigde efficiëntie in de voorgestelde software architectuur: *(i)* de projectie van het array- naar het relationele domein wordt geoptimaliseerd met *array-bewuste vertalingsregels*, met specifieke ondersteuning voor *sparse arrays*; *(ii)* het onderliggende database systeem is toegespitst op het *benutten van moderne hardware*, zoals het voor dit onderzoek gebruikte MonetDB/X100 database systeem [ZBNH05]. Dit databasesysteem voert database vragen uit *per-vector in-cache* om de CPU optimaal te benutten [BZN05], en, het gebruikt een *kolom-georiënteerde datarepresentatie* met, transparante, *lichtgewicht data compressie* om de maximale performance te halen uit de I/O-bandbreedte [ZHNB06].

Curricuum Vitæ

Roberto Cornacchia was born on 10 April 1974 in Rimini, Italy. In 1993, he graduated from secondary school at "Liceo Scientifico Serpieri" in Rimini, Italy. In 1999, he received his Master's degree in computer science from the University of Bologna, Italy, under the supervision of Prof. dr. Paolo Ciaccia, with a dissertation titled "Top Query: Ottimizzazione e Imlementazione in PostgreSQL". From 2000 to 2003, Roberto has worked as an independent IT consultant and co-founded the software house Albini & Fontanot Informatica. In October 2003, he has been hired at the Centrum voor Wiskunde en Informatica (CWI) in Amsterdam, The Netherlands, as project member in the MultimediaN project (www.multimedian.nl). During his Ph.D. research, conducted in the context of the MultimediaN project, Roberto Cornacchia published in several international conferences and journals on databases and information retrieval. In 2007, he received the best student paper award at the European Conference on Information Retrieval (ECIR), for his publication "A parametrised Search System".

Roberto is co-founder of Spinque, where he currently works. Spinque, established in 2010, is spin-off company of CWI that develops search technology for information specialists.

SIKS Dissertation Series

1998-1 Johan van den Akker (CWI). DEGAS - An Active, Temporal Database of Autonomous Objects

1998-2 Floris Wiesman (UM). Information Retrieval by Graphically Browsing Meta-Information

1998-3 Ans Steuten (TUD). A Contribution to the Linguistic Analysis of Business Conversationswithin the Language/Action Perspective

1998-4 Dennis Breuker (UM). Memory versus Search in Games

1998-5 E.W.Oskamp (RUL). Computerondersteuning bij Straftoemeting

1999-1 Mark Sloof (VU). Physiology of Quality Change Modelling; Automated modelling ofQuality Change of Agricultural Products

1999-2 Rob Potharst (EUR). Classification using decision trees and neural nets

1999-3 Don Beal (UM). The Nature of Minimax Search

1999-4 Jacques Penders (UM). The practical Art of Moving Physical Objects

1999-5 Aldo de Moor (KUB). Empowering Communities: A Method for the Legitimate User-DrivenSpecification of Network Information Systems

1999-6 Niek J.E. Wijngaards (VU). Re-design of compositional systems

1999-7 David Spelt (UT). Verification support for object database design

1999-8 Jacques H.J. Lenting (UM). Informed Gambling: Conception and Analysis of a Multi-Agent Mechanismfor Discrete Reallocation.

2000-1 Frank Niessink (VU). Perspectives on Improving Software Maintenance

2000-2 Koen Holtman (TUE). Prototyping of CMS Storage Management

2000-3 Carolien M.T. Metselaar (UvA). Sociaal-organisatorische gevolgen van kennistechnologie;een procesbenadering en actorperspectief.

2000-4 Geert de Haan (VU). ETAG, A Formal Model of Competence Knowledge for User InterfaceDesign

2000-5 Ruud van der Pol (UM). Knowledge-based Query Formulation in Information Retrieval.

2000-6 Rogier van Eijk (UU). Programming Languages for Agent Communication

2000-7 Niels Peek (UU). Decision-theoretic Planning of Clinical Patient Management

2000-8 Veerle Coupe (EUR). Sensitivity Analyis of Decision-Theoretic Networks

2000-9 Florian Waas (CWI). Principles of Probabilistic Query Optimization

2000-10 Niels Nes (CWI). Image Database Management System Design Considerations, Algorithms and Architecture

2000-11 Jonas Karlsson (CWI). Scalable Distributed Data Structures for Database Management

2001-1 Silja Renooij (UU). Qualitative Approaches to Quantifying Probabilistic Networks

2001-2 Koen Hindriks (UU). Agent Programming Languages: Programming with Mental Models

2001-3 Maarten van Someren (UvA). Learning as problem solving

2001-4 Evgueni Smirnov (UM). Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets

2001-5 Jacco van Ossenbruggen (VU). Processing Structured Hypermedia: A Matter of Style

2001-6 Martijn van Welie (VU). Task-based User Interface Design

2001-7 Bastiaan Schonhage (VU). Diva: Architectural Perspectives on Information Visualization

2001-8 Pascal van Eck (VU). A Compositional Semantic Structure for Multi-Agent Systems Dynamics.

2001-9 Pieter Jan 't Hoen (RUL). Towards Distributed Development of Large Object-Oriented Models,Views of Packages as Classes

2001-10 Maarten Sierhuis (UvA). Modeling and Simulating Work PracticeBRAHMS: a multiagent modeling and simulation language forwork practice analysis and design

2001-11 Tom M. van Engers (VUA). Knowledge Management:The Role of Mental Models in Business Systems Design

2002-01 Nico Lassing (VU). Architecture-Level Modifiability Analysis

2002-02 Roelof van Zwol (UT). Modelling and searching web-based document collections

2002-03 Henk Ernst Blok (UT). Database Optimization Aspects for Information Retrieval

2002-04 Juan Roberto Castelo Valdueza (UU). The Discrete Acyclic Digraph Markov Model in Data Mining

2002-05 Radu Serban (VU). The Private Cyberspace Modeling ElectronicEnvironments inhabited by Privacy-concerned Agents

2002-06 Laurens Mommers (UL). Applied legal epistemology; Building a knowledge-based ontology ofthe legal domain

2002-07 Peter Boncz (CWI). Monet: A Next-Generation DBMS Kernel For Query-IntensiveApplications

2002-08 Jaap Gordijn (VU). Value Based Requirements Engineering: Exploring InnovativeE-Commerce Ideas

2002-09 Willem-Jan van den Heuvel(KUB). Integrating Modern Business Applications with Objectified LegacySystems

2002-10 Brian Sheppard (UM). Towards Perfect Play of Scrabble

2002-11 Wouter C.A. Wijngaards (VU). Agent Based Modelling of Dynamics: Biological and Organisational Applications

2002-12 Albrecht Schmidt (Uva). Processing XML in Database Systems

2002-13 Hongjing Wu (TUE). A Reference Architecture for Adaptive Hypermedia Applications

2002-14 Wieke de Vries (UU). Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems

2002-15 Rik Eshuis (UT). Semantics and Verification of UML Activity Diagrams for Workflow Modelling

2002-16 Pieter van Langen (VU). The Anatomy of Design: Foundations, Models and Applications

2002-17 Stefan Manegold (UvA). Understanding, Modeling, and Improving Main-Memory Database Performance

2003-01 Heiner Stuckenschmidt (VU). Ontology-Based Information Sharing in Weakly Structured Environments

2003-02 Jan Broersen (VU). Modal Action Logics for Reasoning About Reactive Systems

2003-03 Martijn Schuemie (TUD). Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy

2003-04 Milan Petkovic (UT). Content-Based Video Retrieval Supported by Database Technology

2003-05 Jos Lehmann (UvA). Causation in Artificial Intelligence and Law - A modelling approach

2003-06 Boris van Schooten (UT). Development and specification of virtual environments

2003-07 Machiel Jansen (UvA). Formal Explorations of Knowledge Intensive Tasks

2003-08 Yongping Ran (UM). Repair Based Scheduling

2003-09 Rens Kortmann (UM). The resolution of visually guided behaviour

2003-10 Andreas Lincke (UvT). Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture

2003-11 Simon Keizer (UT). Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks

2003-12 Roeland Ordelman (UT). Dutch speech recognition in multimedia information retrieval

2003-13 Jeroen Donkers (UM). Nosce Hostem - Searching with Opponent Models

2003-14 Stijn Hoppenbrouwers (KUN). Freezing Language: Conceptualisation Processes across ICT-Supported Organisations

2003-15 Mathijs de Weerdt (TUD). Plan Merging in Multi-Agent Systems

2003-16 Menzo Windhouwer (CWI). Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses

2003-17 David Jansen (UT). Extensions of Statecharts with Probability, Time, and Stochastic Timing

2003-18 Levente Kocsis (UM). Learning Search Decisions

2004-01 Virginia Dignum (UU). A Model for Organizational Interaction: Based on Agents, Founded in Logic

2004-02 Lai Xu (UvT). Monitoring Multi-party Contracts for E-business

2004-03 Perry Groot (VU). A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving

2004-04 Chris van Aart (UvA). Organizational Principles for Multi-Agent Architectures

2004-05 Viara Popova (EUR). Knowledge discovery and monotonicity

2004-06 Bart-Jan Hommes (TUD). The Evaluation of Business Process Modeling Techniques

2004-07 Elise Boltjes (UM). Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes

2004-08 Joop Verbeek(UM). Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politiele gegevensuitwisseling en digitale expertise

2004-09 Martin Caminada (VU). For the Sake of the Argument; explorations into argument-based reasoning

2004-10 Suzanne Kabel (UvA). Knowledge-rich indexing of learning-objects

2004-11 Michel Klein (VU). Change Management for Distributed Ontologies

2004-12 The Duy Bui (UT). Creating emotions and facial expressions for embodied agents

2004-13 Wojciech Jamroga (UT). Using Multiple Models of Reality: On Agents who Know how to Play

2004-14 Paul Harrenstein (UU). Logic in Conflict. Logical Explorations in Strategic Equilibrium

2004-15 Arno Knobbe (UU). Multi-Relational Data Mining

2004-16 Federico Divina (VU). Hybrid Genetic Relational Search for Inductive Learning

2004-17 Mark Winands (UM). Informed Search in Complex Games

2004-18 Vania Bessa Machado (UvA). Supporting the Construction of Qualitative Knowledge Models

2004-19 Thijs Westerveld (UT). Using generative probabilistic models for multimedia retrieval

2004-20 Madelon Evers (Nyenrode). Learning from Design: facilitating multidisciplinary design teams

2005-01 Floor Verdenius (UvA). Methodological Aspects of Designing Induction-Based Applications

2005-02 Erik van der Werf (UM)). AI techniques for the game of Go

2005-03 Franc Grootjen (RUN). A Pragmatic Approach to the Conceptualisation of Language

2005-04 Nirvana Meratnia (UT). Towards Database Support for Moving Object data

2005-05 Gabriel Infante-Lopez (UvA). Two-Level Probabilistic Grammars for Natural Language Parsing

2005-06 Pieter Spronck (UM). Adaptive Game AI

2005-07 Flavius Frasincar (TUE). Hypermedia Presentation Generation for Semantic Web Information Systems

2005-08 Richard Vdovjak (TUE). A Model-driven Approach for Building Distributed Ontology-based Web Applications

2005-09 Jeen Broekstra (VU). Storage, Querying and Inferencing for Semantic Web Languages

2005-10 Anders Bouwer (UvA). Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments

2005-11 Elth Ogston (VU). Agent Based Matchmaking and Clustering - A Decentralized Approach to Search

2005-12 Csaba Boer (EUR). Distributed Simulation in Industry

2005-13 Fred Hamburg (UL). Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen

2005-14 Borys Omelayenko (VU). Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics

2005-15 Tibor Bosse (VU). Analysis of the Dynamics of Cognitive Processes

2005-16 Joris Graaumans (UU). Usability of XML Query Languages

2005-17 Boris Shishkov (TUD). Software Specification Based on Re-usable Business Components

2005-18 Danielle Sent (UU). Test-selection strategies for probabilistic networks

2005-19 Michel van Dartel (UM). Situated Representation

2005-20 Cristina Coteanu (UL). Cyber Consumer Law, State of the Art and Perspectives

2005-21 Wijnand Derks (UT). Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics

2006-01 Samuil Angelov (TUE). Foundations of B2B Electronic Contracting

2006-02 Cristina Chisalita (VU). Contextual issues in the design and use of information technology in organizations

2006-03 Noor Christoph (UvA). The role of metacognitive skills in learning to solve problems

2006-04 Marta Sabou (VU). Building Web Service Ontologies

2006-05 Cees Pierik (UU). Validation Techniques for Object-Oriented Proof Outlines

2006-06 Ziv Baida (VU). Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling

2006-07 Marko Smiljanic (UT). XML schema matching – balancing efficiency and effectiveness by means of clustering

2006-08 Eelco Herder (UT). Forward, Back and Home Again - Analyzing User Behavior on the Web

2006-09 Mohamed Wahdan (UM). Automatic Formulation of the Auditor's Opinion

2006-10 Ronny Siebes (VU). Semantic Routing in Peer-to-Peer Systems

2006-11 Joeri van Ruth (UT). Flattening Queries over Nested Data Types

2006-12 Bert Bongers (VU). Interactivation - Towards an e-cology of people, our technological environment, and the arts

2006-13 Henk-Jan Lebbink (UU). Dialogue and Decision Games for Information Exchanging Agents

2006-14 Johan Hoorn (VU). Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change

2006-15 Rainer Malik (UU). CONAN: Text Mining in the Biomedical Domain

2006-16 Carsten Riggelsen (UU). Approximation Methods for Efficient Learning of Bayesian Networks

2006-17 Stacey Nagata (UU). User Assistance for Multitasking with Interruptions on a Mobile Device

2006-18 Valentin Zhizhkun (UvA). Graph transformation for Natural Language Processing

2006-19 Birna van Riemsdijk (UU). Cognitive Agent Programming: A Semantic Approach

2006-20 Marina Velikova (UvT). Monotone models for prediction in data mining

2006-21 Bas van Gils (RUN). Aptness on the Web

2006-22 Paul de Vrieze (RUN). Fundaments of Adaptive Personalisation

2006-23 Ion Juvina (UU). Development of Cognitive Model for Navigating on the Web

2006-24 Laura Hollink (VU). Semantic Annotation for Retrieval of Visual Resources

2006-25 Madalina Drugan (UU). Conditional log-likelihood MDL and Evolutionary MCMC

2006-26 Vojkan Mihajlovic (UT). Score Region Algebra: A Flexible Framework for Structured Information Retrieval

2006-27 Stefano Bocconi (CWI). Vox Populi: generating video documentaries from semantically annotated media repositories

2006-28 Borkur Sigurbjornsson (UvA). Focused Information Access using XML Element Retrieval

2007-01 Kees Leune (UvT). Access Control and Service-Oriented Architectures

2007-02 Wouter Teepe (RUG). Reconciling Information Exchange and Confidentiality: A Formal Approach

2007-03 Peter Mika (VU). Social Networks and the Semantic Web

2007-04 Jurriaan van Diggelen (UU). Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach

2007-05 Bart Schermer (UL). Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance

2007-06 Gilad Mishne (UvA). Applied Text Analytics for Blogs

2007-07 Natasa Jovanovic' (UT). To Whom It May Concern - Addressee Identification in Face-to-Face Meetings

2007-08 Mark Hoogendoorn (VU). Modeling of Change in Multi-Agent Organizations

2007-09 David Mobach (VU). Agent-Based Mediated Service Negotiation

2007-10 Huib Aldewereld (UU). Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols

2007-11 Natalia Stash (TUE). Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System

2007-12 Marcel van Gerven (RUN). Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty

2007-13 Rutger Rienks (UT). Meetings in Smart Environments; Implications of Progressing Technology

2007-14 Niek Bergboer (UM). Context-Based Image Analysis

2007-15 Joyca Lacroix (UM). NIM: a Situated Computational Memory Model

2007-16 Davide Grossi (UU). Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems

2007-17 Theodore Charitos (UU). Reasoning with Dynamic Networks in Practice

2007-18 Bart Orriens (UvT). On the development an management of adaptive business collaborations

2007-19 David Levy (UM). Intimate relationships with artificial partners

2007-20 Slinger Jansen (UU). Customer Configuration Updating in a Software Supply Network

2007-21 Karianne Vermaas (UU). Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005

2007-22 Zlatko Zlatev (UT). Goal-oriented design of value and process models from patterns

2007-23 Peter Barna (TUE). Specification of Application Logic in Web Information Systems

2007-24 Georgina Ramirez Camps (CWI). Structural Features in XML Retrieval

2007-25 Joost Schalken (VU). Empirical Investigations in Software Process Improvement

2008-01 Katalin Boer-Sorban (EUR). Agent-Based Simulation of Financial Markets: A modular, continuous-time approach

2008-02 Alexei Sharpanskykh (VU). On Computer-Aided Methods for Modeling and Analysis of Organizations

2008-03 Vera Hollink (UvA). Optimizing hierarchical menus: a usage-based approach

2008-04 Ander de Keijzer (UT). Management of Uncertain Data - towards unattended integration

2008-05 Bela Mutschler (UT). Modeling and simulating causal dependencies on process-aware information systems from a cost perspective

2008-06 Arjen Hommersom (RUN). On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective

2008-07 Peter van Rosmalen (OU). Supporting the tutor in the design and support of adaptive e-learning

2008-08 Janneke Bolt (UU). Bayesian Networks: Aspects of Approximate Inference

2008-09 Christof van Nimwegen (UU). The paradox of the guided user: assistance can be counter-effective

2008-10 Wauter Bosma (UT). Discourse oriented summarization

2008-11 Vera Kartseva (VU). Designing Controls for Network Organizations: A Value-Based Approach

2008-12 Jozsef Farkas (RUN). A Semiotically Oriented Cognitive Model of Knowledge Representation

2008-13 Caterina Carraciolo (UvA). Topic Driven Access to Scientific Handbooks

2008-14 Arthur van Bunningen (UT). Context-Aware Querying; Better Answers with Less Effort

2008-15 Martijn van Otterlo (UT). The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains.

2008-16 Henriette van Vugt (VU). Embodied agents from a user's perspective

2008-17 Martin Op 't Land (TUD). Applying Architecture and Ontology to the Splitting and Allying of Enterprises

2008-18 Guido de Croon (UM). Adaptive Active Vision

2008-19 Henning Rode (UT). From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search

2008-20 Rex Arendsen (UvA). Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer met de overheid op de administratieve lasten van bedrijven.

2008-21 Krisztian Balog (UvA). People Search in the Enterprise

2008-22 Henk Koning (UU). Communication of IT-Architecture

2008-23 Stefan Visscher (UU). Bayesian network models for the management of ventilator-associated pneumonia

2008-24 Zharko Aleksovski (VU). Using background knowledge in ontology matching

2008-25 Geert Jonker (UU). Efficient and Equitable Exchange in Air Traffic Management Plan Repair using Spender-signed Currency

2008-26 Marijn Huijbregts (UT). Segmentation, Diarization and Speech Transcription: Surprise Data Unraveled

2008-27 Hubert Vogten (OU). Design and Implementation Strategies for IMS Learning Design

2008-28 Ildiko Flesch (RUN). On the Use of Independence Relations in Bayesian Networks

2008-29 Dennis Reidsma (UT). Annotations and Subjective Machines - Of Annotators, Embodied Agents, Users, and Other Humans

2008-30 Wouter van Atteveldt (VU). Semantic Network Analysis: Techniques for Extracting, Representing and Querying Media Content

2008-31 Loes Braun (UM). Pro-Active Medical Information Retrieval

2008-32 Trung H. Bui (UT). Toward Affective Dialogue Management using Partially Observable Markov Decision Processes

2008-33 Frank Terpstra (UvA). Scientific Workflow Design; theoretical and practical issues

2008-34 Jeroen de Knijf (UU). Studies in Frequent Tree Mining

2008-35 Ben Torben Nielsen (UvT). Dendritic morphologies: function shapes structure

2009-01 Rasa Jurgelenaite (RUN). Symmetric Causal Independence Models

2009-02 Willem Robert van Hage (VU). Evaluating Ontology-Alignment Techniques

2009-03 Hans Stol (UvT). A Framework for Evidence-based Policy Making Using IT

2009-04 Josephine Nabukenya (RUN). Improving the Quality of Organisational Policy Making using Collaboration Engineering

2009-05 Sietse Overbeek (RUN). Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality

2009-06 Muhammad Subianto (UU). Understanding Classification

2009-07 Ronald Poppe (UT). Discriminative Vision-Based Recovery and Recognition of Human Motion

2009-08 Volker Nannen (VU). Evolutionary Agent-Based Policy Analysis in Dynamic Environments

2009-09 Benjamin Kanagwa (RUN). Design, Discovery and Construction of Service-oriented Systems

2009-10 Jan Wielemaker (UvA). Logic programming for knowledge-intensive interactive applications

2009-11 Alexander Boer (UvA). Legal Theory, Sources of Law & the Semantic Web

2009-12 Peter Massuthe (TUE, Humboldt-Universitaet zu Berlin). Operating Guidelines for Services

2009-13 Steven de Jong (UM). Fairness in Multi-Agent Systems

2009-14 Maksym Korotkiy (VU). From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA).

2009-15 Rinke Hoekstra (UvA). Ontology Representation - Design Patterns and Ontologies that Make Sense

2009-16 Fritz Reul (UvT). New Architectures in Computer Chess

2009-17 Laurens van der Maaten (UvT). Feature Extraction from Visual Data

2009-18 Fabian Groffen (CWI). Armada, An Evolving Database System

2009-19 Valentin Robu (CWI). Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets

2009-20 Bob van der Vecht (UU). Adjustable Autonomy: Controling Influences on Decision Making

2009-21 Stijn Vanderlooy (UM). Ranking and Reliable Classification

2009-22 Pavel Serdyukov (UT). Search For Expertise: Going beyond direct evidence

2009-23 Peter Hofgesang (VU). Modelling Web Usage in a Changing Environment

2009-24 Annerieke Heuvelink (VUA). Cognitive Models for Training Simulations

2009-25 Alex van Ballegooij (CWI). "RAM: Array Database Management through Relational Mapping"

2009-26 Fernando Koch (UU). An Agent-Based Model for the Development of Intelligent Mobile Services

2009-27 Christian Glahn (OU). Contextual Support of social Engagement and Reflection on the Web

2009-28 Sander Evers (UT). Sensor Data Management with Probabilistic Models

2009-29 Stanislav Pokraev (UT). Model-Driven Semantic Integration of Service-Oriented Applications

2009-30 Marcin Zukowski (CWI). Balancing vectorized query execution with bandwidth-optimized storage

2009-31 Sofiya Katrenko (UvA). A Closer Look at Learning Relations from Text

2009-32 Rik Farenhorst (VU) and Remco de Boer (VU). Architectural Knowledge Management: Supporting Architects and Auditors

2009-33 Khiet Truong (UT). How Does Real Affect Affect Affect Recognition In Speech?

2009-34 Inge van de Weerd (UU). Advancing in Software Product Management: An Incremental Method Engineering Approach

2009-35 Wouter Koelewijn (UL). Privacy en Politiegegevens; Over geautomatiseerde normatieve informatie-uitwisseling

2009-36 Marco Kalz (OUN). Placement Support for Learners in Learning Networks

2009-37 Hendrik Drachsler (OUN). Navigation Support for Learners in Informal Learning Networks

2009-38 Riina Vuorikari (OU). Tags and self-organisation: a metadata ecology for learning resources in a multilingual context

2009-39 Christian Stahl (TUE, Humboldt-Universitaet zu Berlin). Service Substitution – A Behavioral Approach Based on Petri Nets

2009-40 Stephan Raaijmakers (UvT). Multinomial Language Learning: Investigations into the Geometry of Language

2009-41 Igor Berezhnyy (UvT). Digital Analysis of Paintings

2009-42 Toine Bogers (UvT). Recommender Systems for Social Bookmarking

2009-43 Virginia Nunes Leal Franqueira (UT). Finding Multi-step Attacks in Computer Networks using Heuristic Search and Mobile Ambients

2009-44 Roberto Santana Tapia (UT). Assessing Business-IT Alignment in Networked Organizations

2009-45 Jilles Vreeken (UU). Making Pattern Mining Useful

2009-46 Loredana Afanasiev (UvA). Querying XML: Benchmarks and Recursion

2010-01 Matthijs van Leeuwen (UU). Patterns that Matter

2010-02 Ingo Wassink (UT). Work flows in Life Science

2010-03 Joost Geurts (CWI). A Document Engineering Model and Processing Framework for Multimedia documents

2010-04 Olga Kulyk (UT). Do You Know What I Know? Situational Awareness of Co-located Teams in Multidisplay Environments

2010-05 Claudia Hauff (UT). Predicting the Effectiveness of Queries and Retrieval Systems

2010-06 Sander Bakkes (UvT). Rapid Adaptation of Video Game AI

2010-07 Wim Fikkert (UT). Gesture interaction at a Distance

2010-08 Krzysztof Siewicz (UL). Towards an Improved Regulatory Framework of Free Software. Protecting user freedoms in a world of software communities and eGovernments

2010-09 Hugo Kielman (UL). A Politiele gegevensverwerking en Privacy, Naar een effectieve waarborging

2010-10 Rebecca Ong (UL). Mobile Communication and Protection of Children

2010-11 Adriaan Ter Mors (TUD). The world according to MARP: Multi-Agent Route Planning

2010-12 Susan van den Braak (UU). Sensemaking software for crime analysis

2010-13 Gianluigi Folino (RUN). High Performance Data Mining using Bio-inspired techniques

2010-14 Sander van Splunter (VU). Automated Web Service Reconfiguration

2010-15 Lianne Bodenstaff (UT). Managing Dependency Relations in Inter-Organizational Models

2010-16 Sicco Verwer (TUD). Efficient Identification of Timed Automata, theory and practice

2010-17 Spyros Kotoulas (VU). Scalable Discovery of Networked Resources: Algorithms, Infrastructure, Applications

2010-18 Charlotte Gerritsen (VU). Caught in the Act: Investigating Crime by Agent-Based Simulation

2010-19 Henriette Cramer (UvA). People's Responses to Autonomous and Adaptive Systems

2010-20 Ivo Swartjes (UT). Whose Story Is It Anyway? How Improv Informs Agency and Authorship of Emergent Narrative

2010-21 Harold van Heerde (UT). Privacy-aware data management by means of data degradation

2010-22 Michiel Hildebrand (CWI). End-user Support for Access to
Heterogeneous Linked Data

2010-23 Bas Steunebrink (UU). The Logical Structure of Emotions

2010-24 Dmytro Tykhonov Designing Generic and Efficient Negotiation Strategies

2010-25 Zulfiqar Ali Memon (VU). Modelling Human-Awareness for Ambient Agents: A Human Mindreading Perspective

2010-26 Ying Zhang (CWI). XRPC: Efficient Distributed Query Processing on Heterogeneous XQuery Engines

2010-27 Marten Voulon (UL). Automatisch contracteren

2010-28 Arne Koopman (UU). Characteristic Relational Patterns

2010-29 Stratos Idreos(CWI). Database Cracking: Towards Auto-tuning Database Kernels

2010-30 Marieke van Erp (UvT). Accessing Natural History - Discoveries in data cleaning, structuring, and retrieval

2010-31 Victor de Boer (UvA). Ontology Enrichment from Heterogeneous Sources on the Web

2010-32 Marcel Hiel (UvT). An Adaptive Service Oriented Architecture: Automatically solving Interoperability Problems

2010-33 Robin Aly (UT). Modeling Representation Uncertainty in Concept-Based Multimedia Retrieval

2010-34 Teduh Dirgahayu (UT). Interaction Design in Service Compositions

2010-35 Dolf Trieschnigg (UT). Proof of Concept: Concept-based Biomedical Information Retrieval

2010-36 Jose Janssen (OU). Paving the Way for Lifelong Learning; Facilitating competence development through a learning path specification

2010-37 Niels Lohmann (TUE). Correctness of services and their composition

2010-38 Dirk Fahland (TUE). From Scenarios to components

2010-39	Ghazanfar Farooq Siddiqui (VU). Integrative modeling of emotions in virtual agents
2010-40	Mark van Assem (VU). Converting and Integrating Vocabularies for the Semantic Web
2010-41	Guillaume Chaslot (UM). Monte-Carlo Tree Search
2010-42	Sybren de Kinderen (VU). Needs-driven service bundling in a multi-supplier setting - the computational e3-service approach
2010-43	Peter van Kranenburg (UU). A Computational Approach to Content-Based Retrieval of Folk Song Melodies
2010-44	Pieter Bellekens (TUE). An Approach towards Context-sensitive and User-adapted Access to Heterogeneous Data Sources, Illustrated in the Television Domain
2010-45	Vasilios Andrikopoulos (UvT). A theory and model for the evolution of software services
2010-46	Vincent Pijpers (VU). e3alignment: Exploring Inter-Organizational Business-ICT Alignment
2010-47	Chen Li (UT). Mining Process Model Variants: Challenges, Techniques, Examples
2010-48	Withdrawn
2010-49	Jahn-Takeshi Saito (UM). Solving difficult game positions
2010-50	Bouke Huurnink (UvA). Search in Audiovisual Broadcast Archives
2010-51	Alia Khairia Amin (CWI). Understanding and supporting information seeking tasks in multiple sources
2010-52	Peter-Paul van Maanen (VU). Adaptive Support for Human-Computer Teams: Exploring the Use of Cognitive Models of Trust and Attention
2010-53	Edgar Meij (UvA). Combining Concepts and Language Models for Information Access
2011-01	Botond Cseke (RUN). Variational Algorithms for Bayesian Inference in Latent Gaussian Models
2011-02	Nick Tinnemeier(UU). Organizing Agent Organizations. Syntax and Operational Semantics of an Organization-Oriented Programming Language
2011-03	Jan Martijn van der Werf (TUE). Compositional Design and Verification of Component-Based Information Systems
2011-04	Hado van Hasselt (UU). Insights in Reinforcement Learning; Formal analysis and empirical evaluation of temporal-differencelearning algorithms
2011-05	Base van der Raadt (VU). Enterprise Architecture Coming of Age - Increasing the Performance of an Emerging Discipline.
2011-06	Yiwen Wang (TUE). Semantically-Enhanced Recommendations in Cultural Heritage
2011-07	Yujia Cao (UT). Multimodal Information Presentation for High Load Human Computer Interaction
2011-08	Nieske Vergunst (UU). BDI-based Generation of Robust Task-Oriented Dialogues
2011-09	Tim de Jong (OU). Contextualised Mobile Media for Learning
2011-10	Bart Bogaert (UvT). Cloud Content Contention
2011-11	Dhaval Vyas (UT). Designing for Awareness: An Experience-focused HCI Perspective
2011-12	Carmen Bratosin (TUE). Grid Architecture for Distributed Process Mining
2011-13	Xiaoyu Mao (UvT). Airport under Control. Multiagent Scheduling for Airport Ground Handling
2011-14	Milan Lovric (EUR). Behavioral Finance and Agent-Based Artificial Markets
2011-15	Marijn Koolen (UvA). The Meaning of Structure: the Value of Link Evidence for Information Retrieval
2011-16	Maarten Schadd (UM). Selective Search in Games of Different Complexity

2011-17 Jiyin He (UvA). Exploring Topic Structure: Coherence, Diversity and Relatedness

2011-18 Mark Ponsen (UM). Strategic Decision-Making in complex games

2011-19 Ellen Rusman (OU). The Mind ' s Eye on Personal Profiles

2011-20 Qing Gu (VU). Guiding service-oriented software engineering - A view-based approach

2011-21 Linda Terlouw (TUD). Modularization and Specification of Service-Oriented Systems

2011-22 Junte Zhang (UvA). System Evaluation of Archival Description and Access

2011-23 Wouter Weerkamp (UvA). Finding People and their Utterances in Social Media

2011-24 Herwin van Welbergen (UT). Behavior Generation for Interpersonal Coordination with Virtual Humans On Specifying, Scheduling and Realizing Multimodal Virtual Human Behavior

2011-25 Syed Waqar ul Qounain Jaffry (VU)). Analysis and Validation of Models for Trust Dynamics

2011-26 Matthijs Aart Pontier (VU). Virtual Agents for Human Communication - Emotion Regulation and Involvement-Distance Trade-Offs in Embodied Conversational Agents and Robots

2011-27 Aniel Bhulai (VU). Dynamic website optimization through autonomous management of design patterns

2011-28 Rianne Kaptein(UvA). Effective Focused Retrieval by Exploiting Query Context and Document Structure

2011-29 Faisal Kamiran (TUE). Discrimination-aware Classification

2011-30 Egon van den Broek (UT). Affective Signal Processing (ASP): Unraveling the mystery of emotions

2011-31 Ludo Waltman (EUR). Computational and Game-Theoretic Approaches for Modeling Bounded Rationality

2011-32 Nees-Jan van Eck (EUR). Methodological Advances in Bibliometric Mapping of Science

2011-33 Tom van der Weide (UU). Arguing to Motivate Decisions

2011-34 Paolo Turrini (UU). Strategic Reasoning in Interdependence: Logical and Game-theoretical Investigations

2011-35 Maaike Harbers (UU). Explaining Agent Behavior in Virtual Training

2011-36 Erik van der Spek (UU). Experiments in serious game design: a cognitive approach

2011-37 Adriana Burlutiu (RUN). Machine Learning for Pairwise Data, Applications for Preference Learning and Supervised Network Inference

2011-38 Nyree Lemmens (UM). Bee-inspired Distributed Optimization

2011-39 Joost Westra (UU). Organizing Adaptation using Agents in Serious Games

2011-40 Viktor Clerc (VU). Architectural Knowledge Management in Global Software Development

2011-41 Luan Ibraimi (UT). Cryptographically Enforced Distributed Data Access Control

2011-42 Michal Sindlar (UU). Explaining Behavior through Mental State Attribution

2011-43 Henk van der Schuur (UU). Process Improvement through Software Operation Knowledge

2011-44 Boris Reuderink (UT). Robust Brain-Computer Interfaces

2011-45 Herman Stehouwer (UvT). Statistical Language Models for Alternative Sequence Selection

2011-46 Beibei Hu (TUD). Towards Contextualized Information Delivery: A Rule-based Architecture for the Domain of Mobile Police Work

2011-47 Azizi Bin Ab Aziz(VU). Exploring Computational Models for Intelligent Support of Persons with Depression

2011-48 Mark Ter Maat (UT). Response Selection and Turn-taking for a Sensitive Artificial Listening Agent

2011-49 Andreea Niculescu (UT). Conversational interfaces for task-oriented spoken dialogues: design aspects influencing interaction quality

2012-01 Terry Kakeeto (UvT). Relationship Marketing for SMEs in Uganda

2012-02 Muhammad Umair(VU). Adaptivity, emotion, and Rationality in Human and Ambient Agent Models

2012-03 Adam Vanya (VU). Supporting Architecture Evolution by Mining Software Repositories

2012-04 Jurriaan Souer (UU). Development of Content Management System-based Web Applications

2012-05 Marijn Plomp (UU). Maturing Interorganisational Information Systems

2012-06 Wolfgang Reinhardt (OU). Awareness Support for Knowledge Workers in Research Networks

2012-07 Rianne van Lambalgen (VU). When the Going Gets Tough: Exploring Agent-based Models of Human Performance under Demanding Conditions

2012-08 Gerben de Vries (UvA). Kernel Methods for Vessel Traject

2012-09 Ricardo Neisse (UT). Trust and Privacy Management Support for Context-Aware Service Platforms

2012-10 David Smits (TUE). Towards a Generic Distributed Adaptive Hypermedia Environment

2012-11 J.C.B. Rantham Prabhakara (TUE). Process Mining in the Large: Preprocessing, Discovery, and Diagnostics

2012-12 Kees van der Sluijs (TUE). Model Driven Design and Data Integration in Semantic Web Information Systems

2012-13 Suleman Shahid (UvT). Fun and Face: Exploring non-verbal expressions of emotion during playful interactions

2012-14 Evgeny Knutov(TUE). Generic Adaptation Framework for Unifying Adaptive Web-based Systems

2012-15 Natalie van der Wal (VU). Social Agents. Agent-Based Modelling of Integrated Internal and Social Dynamics of Cognitive and Affective Processes

2012-16 Fiemke Both (VU). Helping people by understanding them - Ambient Agents supporting task execution and depression treatment

2012-17 Amal Elgammal (UvT). Towards a Comprehensive Framework for Business Process Compliance

2012-18 Eltjo Poort (VU). Improving Solution Architecting Practices

2012-19 Helen Schonenberg (TUE). What's Next? Operational Support for Business Process Execution

2012-20 Ali Bahramisharif (RUN). Covert Visual Spatial Attention, a Robust Paradigm for Brain-Computer Interfacing