UNIVERSITÉ PARIS-SUD

ÉCOLE DOCTORALE INFORMATIQUE DE PARIS-SUD (ED 427)

Laboratoire de Recherche en Informatique (LRI)

***DISCIPLINE** INFORMATIQUE*

THÈSE DE DOCTORAT

presentée en vue d'obtention du titre de docteur le 25/09/2014

par **Jesús CAMACHO RODRÍGUEZ**

Efficient techniques for large-scale Web data management

Thèse dirigée par :

Dario Colazzo	Université Paris-Dauphine
Ioana Manolescu	Inria and Université Paris-Sud

Rapporteurs :

Donald Kossmann	ETH Zürich, Switzerland
Volker Markl	TU Berlin, Germany

Examinateurs :

Reza Akbarinia	Inria and Université Montpellier II
Marc Baboulin	Université Paris-Sud and Inria
Philippe Rigaux	Conservatoire National des Arts et Métiers

To my parents, for their everlasting love,
support and encouragement.

Abstract

"Efficient techniques for large-scale Web data management"

Jesús Camacho Rodríguez

The recent development of commercial cloud computing environments has strongly impacted research and development in distributed software platforms. Cloud providers offer a distributed, shared-nothing infrastructure, that may be used for data storage and processing.

In parallel with the development of cloud platforms, programming models that seamlessly parallelize the execution of data-intensive tasks over large clusters of commodity machines have received significant attention, starting with the MapReduce model very well known by now, and continuing through other novel and more expressive frameworks. As these models are increasingly used to express analytical-style data processing tasks, the need for higher-level languages that ease the burden of writing complex queries for these systems arises.

This thesis investigates the efficient management of Web data on large-scale infrastructures. In particular, we study the performance and cost of exploiting cloud services to build Web data warehouses, and the parallelization and optimization of query languages that are tailored towards querying Web data declaratively.

First, we present AMADA, an architecture for warehousing large-scale Web data in commercial cloud platforms. AMADA operates in a Software as a Service (SaaS) approach, allowing users to upload, store, and query large volumes of Web data. Since cloud users support monetary costs directly connected to their consumption of resources, our focus is not only on query performance from an execution time perspective, but also on the monetary costs associated to this processing. In particular, we study the applicability of several content indexing strategies, and show that they lead not only to reducing query evaluation time, but also, importantly, to reducing the monetary costs associated with the exploitation of the cloud-based warehouse.

Second, we consider the efficient parallelization of the execution of complex queries over XML documents, implemented within our system PAXQuery. We provide novel algorithms showing how to translate such queries into plans expressed in the PArallelization ConTracts (PACT) programming model. These plans are then optimized and executed in parallel by the Stratosphere system. We demonstrate the efficiency and scalability of our approach through experiments on hundreds of GB of XML data.

Finally, we present a novel approach for identifying and reusing common subexpressions occurring in Pig Latin scripts. In particular, we lay the foundation of our reuse-based algorithms by formalizing the semantics of the Pig Latin query language with extended nested relational algebra for bags. Our algorithm, named PigReuse, operates on the algebraic representations of Pig Latin scripts, identifies subexpression merging opportunities, selects the best ones to execute based on a cost function, and

merges other equivalent expressions to share its result. We bring several extensions to the algorithm to improve its performance. Our experiment results demonstrate the efficiency and effectiveness of our reuse-based algorithms and optimization strategies.

Keywords: Web data, XML, commercial cloud services, indexing strategies, query processing, distributed storage, query parallelization, XQuery, multi-query optimization, Pig Latin.

Résumé

"Techniques efficaces de gestion de données Web à grande échelle"

Jesús Camacho Rodríguez

Le développement récent des offres commerciales autour du cloud computing a fortement influé sur la recherche et le développement des plateformes de distribution numérique. Les fournisseurs du cloud offrent une infrastructure de distribution extensible qui peut être utilisée pour le stockage et le traitement des données.

En parallèle avec le développement des plates-formes de cloud computing, les modèles de programmation qui parallélisent de manière transparente l'exécution des tâches gourmandes en données sur des machines standards ont suscité un intérêt considérable, à commencer par le modèle MapReduce très connu aujourd'hui puis par d'autres frameworks plus récents et complets. Puisque ces modèles sont de plus en plus utilisés pour exprimer les tâches de traitement de données analytiques, la nécessité se fait ressentir dans l'utilisation des langages de haut niveau qui facilitent la charge de l'écriture des requêtes complexes pour ces systèmes.

Cette thèse porte sur des modèles et techniques d'optimisation pour le traitement efficace de grandes masses de données du Web sur des infrastructures à grande échelle. Plus particulièrement, nous étudions la performance et le coût d'exploitation des services de cloud computing pour construire des entrepôts de données Web ainsi que la parallélisation et l'optimisation des langages de requêtes conçus sur mesure selon les données déclaratives du Web.

Tout d'abord, nous présentons AMADA, une architecture d'entreposage de données Web à grande échelle dans les plateformes commerciales de cloud computing. AMADA opère comme logiciel en tant que service, permettant aux utilisateurs de télécharger, stocker et interroger de grands volumes de données Web. Sachant que les utilisateurs du cloud prennent en charge les coûts monétaires directement liés à leur consommation de ressources, notre objectif n'est pas seulement la minimisation du temps d'exécution des requêtes, mais aussi la minimisation des coûts financiers associés aux traitements de données. Plus précisément, nous étudions l'applicabilité de plusieurs stratégies d'indexation de contenus et nous montrons qu'elles permettent non seulement de réduire le temps d'exécution des requêtes mais aussi, et surtout, de diminuer les coûts monétaires liés à l'exploitation de l'entrepôt basé sur le cloud.

Ensuite, nous étudions la parallélisation efficace de l'exécution de requêtes complexes sur des documents XML mis en œuvre au sein de notre système PAXQuery. Nous fournissons de nouveaux algorithmes montrant comment traduire ces requêtes dans des plans exprimés par le modèle de programmation PACT (PArallelization Contracts). Ces plans sont ensuite optimisés et exécutés en parallèle par le système Stratosphere. Nous démontrons l'efficacité et l'extensibilité de notre approche à travers des expérimentations sur des centaines de Go de données XML.

Enfin, nous présentons une nouvelle approche pour l'identification et la réutilisa-

tion des sous-expressions communes qui surviennent dans les scripts Pig Latin. Notre algorithme, nommé PigReuse, agit sur les représentations algébriques des scripts Pig Latin, identifie les possibilités de fusion des sous-expressions, sélectionne les meilleurs à exécuter en fonction du coût et fusionne d'autres expressions équivalentes pour partager leurs résultats. Nous apportons plusieurs extensions à l'algorithme afin d'améliorer sa performance. Nos résultats expérimentaux démontrent l'efficacité et la rapidité de nos algorithmes basés sur la réutilisation et des stratégies d'optimisation.

Mot-clés: données Web, XML, plateformes commerciales de cloud computing, stratégies d'indexation, traitement des requêtes, entreposage distribué, parallélisation de l'exécution de requêtes, XQuery, optimisation multi-requête, Pig Latin.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The volume and the rate at which data is being created are increasing very rapidly ever since the inception of the World Wide Web. Data-rich Web sites such as product catalogs, social media sites, RSS and tweets, blogs and online publications exemplify this trend. Further, Web data is very heterogeneous and is produced in many different (possibly complex) formats.

Consequently, an increasing part of the world's interesting data is either shared through the Web, or directly produced through and for Web platforms. By today, many organizations recognize the value of the trove of Web data. However, the storage and processing of those big volumes of heterogeneous (structured, semi-structured and unstructured) data poses a series of challenges, concerning: the right models for describing heterogeneous, complex-structure, distributed data; the languages to be used to expressively manipulate Web data; finally, the architectures and concrete algorithms to be put to task in order to efficiently implement the chosen languages.

1.1 Cloud and massive parallelism for Web data

The work undertaken in this thesis is placed in the perspective of taking advantage of *cloud* infrastructures and *massive parallelism* software frameworks in order to address the above challenges. We motivate these choices below before outlining our contributions.

The recent development of commercial cloud computing environments [AWS, GCP, WA] has strongly impacted research and development in distributed software platforms. Cloud providers offer a distributed, shared-nothing infrastructure, that may be used for data storage and processing. *In this thesis, we explore efficient Web data management architectures built on top of these platforms*, and study the performance of the resulting Web data warehouse platforms. Moreover, resources consumption is directly translated into monetary costs for the user running the warehouse, so it becomes necessary to study the interplay between classical data management techniques, and in particular content indexing, and these costs.

In parallel with the development of cloud platforms, programming models that seamlessly parallelize the execution of data-intensive tasks over large clusters of com-

modity machines have received significant attention, starting with the MapReduce model [DG04] very well known by now, and continuing through many novel and more expressive frameworks such as [BEH+10, ZCD+12]. As these models are increasingly used to express analytical-style data processing tasks, the need for higher-level languages that ease the burden of writing complex queries for these systems arises. *In this thesis, we investigate efficient techniques to translate complex operations expressed in these high-level languages into implicit parallel programming models* such as those referred to above. Further, *we consider optimization opportunities associated to these new systems*.

1.2 Contributions and organization of the thesis

Aiming to study efficient techniques for large-scale Web data management, this thesis addresses three different main problems: the management of Web data using commercial cloud services, the parallelization of XQuery processing over large-scale infrastructures, and the reuse-based optimization of Pig Latin queries. In the following, we provide an overview of the organization of the thesis and we outline our main contributions.

Chapter 2 introduces the scientific background necessary in present the contributions of our work, as well as the main relevant scientific results in the area.

Chapter 3 presents AMADA, an architecture for warehousing Web data using commercial cloud services. The contributions of this chapter are the following:

- We present a generic architecture for large-scale warehousing of complex Web data using commercial cloud platforms. Particular attention is devoted to modeling the monetary costs associated to the exploitation of the warehouse.
- We investigate the usage of content indexing for tree-shaped data (in particular, XML data). In AMADA, indexes serve as a tool to both improve query performance, and reduce the warehouse total monetary costs.
- We describe a concrete implementation of our architecture on top of the Amazon Web Services (AWS, in short) platform, among the most widely adopted commercial cloud platform nowadays. We show that indexing can reduce processing time by up to two orders of magnitude and costs by one order of magnitude; moreover, index creation costs amortize very quickly as more queries are run. Since there is a strong similarity among commercial cloud platforms, our results could easily carry on to another platform.

Chapter 4 presents PAXQuery, a massively parallel processor of XML queries. The contributions of this chapter are the following:

- We present a novel methodology for massively parallel evaluation of XQuery which builds on the *PArallelization ConTracts* (PACT, in short) model [BEH+10], as well as previous research in algebraic XQuery representation and optimization.
- We provide translation algorithms from the algebraic operators required by a

large fragment of XQuery into PACT operators. In contrast to previous work, this enables parallel XQuery evaluation without requiring data or query partitioning effort from the application.
- We model the translation of complex flavors of join operators (including nested and/or outer joins, disjunctive joins etc.) into PACT. The interest of this translation goes beyond the context of XQuery evaluation, as it can be adopted to compile programs expressed in other high-level languages into PACT, thus paving the way to their efficient, parallel execution.
- We fully implemented our translation technique into our PAXQuery platform. Our experimental evaluation demonstrates that our translation approach effectively parallelizes XQuery evaluation scaling well beyond competitor approaches, in particular for what concerns queries featuring joins across different documents.

Chapter 5 considers the problem of identifying and reusing common sub-expressions occurring in Pig Latin [ORS+08] scripts. The contributions of this chapter are the following:
- We formalize the representation of Pig Latin scripts based on an existing well-established algebraic formalism, specifically Nested Relational Algebra for Bags (NRAB) [GM93]. This provides a formal foundation for accurately identifying common expressions in batches of Pig Latin scripts.
- We propose PigReuse, a multi-query optimization algorithm that merges equivalent sub-expressions it identifies in directed acyclic graphs of NRAB operators corresponding to PigLatin scripts. After identifying such reutilization opportunities, PigReuse produces an optimal merged plan where redundant computations have been eliminated. PigReuse relies on an efficient Binary Integer Linear Programming solver to select the best plan based on the cost function provided.
- We present extensions to our baseline PigReuse optimization algorithm to improve its *effectiveness*, i.e., increase the number of common subexpressions it detects.
- We present an initial experimental evaluation of our techniques. At the time of this writing, the work continues within our group.

Chapter 6 concludes and outlines possible future directions.

Chapter 2

State of the Art

This chapter presents the background needed by the presentation of the research work performed in the thesis. The chapter organization derives from the following considerations on the problem of (massively parallel) Web data management:

- One of the main characteristics of Web data is its heterogeneity, i.e., the varying formats in which data is generated and consumed. In particular, tree-structured formats, such as XML or JSON, have gained popularity and wide adoption over the years. Section 2.1 recalls the essential features of the XML format as well as the main research results in the area of XML data management, that are directly relevant for the thesis.
- The notions of query containment and equivalence have been extensively studied by the research community. They are the basis for many important classical optimization works that focus on problems such as multi-query optimization or view selection. Section 2.2 provides a brief recall of the main results in this area.
- The management of data in the cloud has been the focus of attention of many researchers for the last few years, with studies around topics such as elasticity, multi-tenancy, services pricing, etc. We briefly present the current state of the art in Section 2.3.
- We overview parallel data processing frameworks and the opportunities they bring towards tackling the challenges posed by Web-scale data storage and processing, in Section 2.4.

2.1 The Extensible Markup Language (XML)

The Extensible Markup Language (XML) [W3C08] is a W3C standard used widely for data exchange over the Web. It defines an easily readable semi-structured data model that is generic and platform-independent. Moreover, it is a suitable format for integrating data that do not abide by a strict schema.

2.1.1 XML data model

We view XML data as a forest of ordered, node-labeled, unranked trees, as outlined by the simple grammar:

$$\begin{array}{lrcl} \textsf{Element} & e & ::= & \underline{\mathsf{l}_i}(\underline{\mathsf{a}_j} \ldots \underline{\mathsf{a}_k}) \\ \textsf{Tree} & d & ::= & \underline{\mathsf{s}_i} \mid \overline{e}[f] \\ \textsf{Forest} & f & ::= & () \mid f,f \mid d \end{array}$$

An element node has a label $\underline{\mathsf{l}_i}$ and a (possibly empty) sequence of attributes $(\underline{\mathsf{a}_j} \ldots \underline{\mathsf{a}_k})$. In turn, a tree d is either a text node $\underline{\mathsf{s}_i}$, or an element node e having a forest of children. In accordance with the W3C's XML data model, each node is endowed with a unique identity, which we materialize through the $_{i,j,k}$ indices. Finally, a forest f is a sequence of XML trees; $()$ denotes the empty forest.

2.1.2 XML query languages

Over the years, multiple query languages have been proposed for navigating and querying XML documents.

The most widely used and supported XML languages are XPath [W3C14a] and XQuery [W3C14b], W3C standards whose recommendation has been updated recently, as of April 2014. The current versions of both languages rely on a common data model, namely XDM [W3C14c]. In the following, we provide a brief description of these languages.

The formal description of the XML dialects supported in our work is delegated to the subsequent chapters, as the dialects used in AMADA (Chapter 3) and PAXQuery (Chapter 4) queries are different.

XPath is an expression language that allows navigation in XML documents to select a sequence of nodes and values as described by the data model. The name of the language derives from its most distinctive feature, the *path expression*, which provides a means of navigating through the hierarchical structure of an XML document. The current version of XPath is XPath 3.0; the language is a syntactic subset of XQuery (described below).

XQuery is a functional language designed to write queries that are concise and easily understood. It is also flexible enough to query a broad spectrum of XML information sources, including both databases and documents.

XQuery is more powerful than XPath, and actually makes use of XPath to access specific parts of the XML documents. The core of XQuery are *FLWOR* expressions, where:

- *For* clauses bind each node of a sequence to a variable.
- *Let* clauses bind a sequence to a variable.
- *Where* clauses filter the nodes on a boolean expression.
- *Order* clauses sort (node and value) lists according to a specific order.
- *Return* clauses determine the output of the expression and is capable of constructing new XML data.

Various *built-in* functions (including aggregate functions such as min, max etc.), as well as *user defined functions* (UDFs) can be used in XQuery expressions, turning it to a Turing-complete language.

2.1.3 XML data management

Along with the proliferation of available XML data, there was an increasing interest in the data management community on methods for storing and querying efficiently large amounts of XML data. We describe the most prominent efforts below.

2.1.3.1 Centralized systems

Many works have focused on employing relational database systems (RDBMSs, in short) for the storage of XML documents. These works proposed techniques to map the XML documents to relational tables [DFS99, FK99a, TVB+02, ZPR02, DTCO03]. Then, most of these systems translate XML queries into relational ones in order to retrieve the data.

The most important commercial RDBMSs also provide support for XML, including IBM DB2 [BCJ+05], Microsoft SQL Server [PCS+04] and Oracle Database [MLK+05]. Furthermore, the SQL/XML standard [ISO03] is an extension to the SQL language introduced in 2003, providing the *xml* native datatype, which can be used to store and retrieve XML documents.

Along with relational approaches, query processing in centralized settings has been thoroughly studied through XML native formalisms [PWLJ04, DPX04, BGvK+06, MHM06, RSF06, MPV09]. These works rely on algebras that decompose the processing of a query into *operators*, such as: *navigation* (or *tree pattern matching*), which given a path (or tree pattern) query, extracts from a document *tuples* of nodes matching it; *selection*; *projection*; *join* etc.

Further, several open-source XML database systems and XQuery processors have emerged over the last few years. Widely used examples include BaseX [Bas], Saxon [Sax], and Qizx [Qiz].

Finally, classical data management optimization problems have also been extensively studied in the specific XML context. Examples include view-based query rewriting [MFK01, BOB+04, ODPC06, ABMP07a, TYÖ+08, MKVZ11], view selection [MS05, TYÖ+08, TYT+09, KMV12], or view maintenance [BC10, CGM11, BTCU12, BGMS13].

2.1.3.2 Distributed systems

The problem of distributed XML data management has been previously addressed from many angles.

The Xyleme project [ACFR02] built a distributed XML data warehouse prototype, and explored in this setting problems related to query optimization, change control or data integration.

[KOD10] studied the vertical partitioning of XML databases and associated optimization techniques to efficiently execute queries in distributed systems.

The popularity of peer-to-peer overlay networks in the decade of the 2000s led to the emergence of many works that focused on the problem of answering queries over P2P XML databases. Some works in this area are described in [KP05, BC06, AMP+08, KKMZ12].

2.2 Containment and equivalence-based optimizations

The notions of query containment and query equivalence are the basis for highly related and largely studied problems such as multi-query optimization, view-based query rewriting, and view selection.

2.2.1 Containment and equivalence

Given a query q and a database $\mathscr{D}$, let $q(\mathscr{D})$ denote the result of evaluating q over $\mathscr{D}$. Given two queries q_1, q_2, we say that q_1 is contained *in q_2 if $q_1(\mathscr{D}) \subseteq q_2(\mathscr{D})$ for any database $\mathscr{D}$. Further, q_1 and q_2 are* equivalent *if and only if $q_1(\mathscr{D}) \equiv q_2(\mathscr{D})$ for any database $\mathscr{D}$.*

Query containment and equivalence, that have been extensively theoretically studied [CM77, ASU79a, ASU79b, JK83, CV93, LW97, CR00], enable the comparison between different formulations of queries, and are crucial in reuse-based optimization problems.

2.2.2 Multi-query optimization

Given a query workload $\mathscr{Q}$, multi-query optimization (MQO) *is the problem of optimizing $\mathscr{Q}$ by identifying common sub-expressions, which can be evaluated only once and be re-used to speed up the evaluation.*

MQO has been the focus of research for many years. Early works [PS88, Sel88] focused on expensive exhaustive algorithms and the solutions were not integrated with existing system optimizers.

[RSSB00] was the first to integrate MQO into a Volcano-style optimizer, while [ZLFL07] presents a completely integrated MQO solution that comprises the maintenance and explotation of materialized views too. Finally, the recent [SLZ12] presents a MQO approach that takes into account physical requirements (e.g., data partitioning) of the consumers of common sub-expressions in order to propose globally optimal execution plans.

2.2.3 View-based optimizations

As with MQO, the foundation of view-based optimization is query containment and equivalence. View-based optimization problems have received significant atten-

tion by the research community because of their important potential for improving query evaluation performance. We briefly recall the main relevant notions below.

Equivalent query rewriting. *Given a query q and a set of views $\mathcal{V}$, an* equivalent rewriting of q using $\mathcal{V}$ *is an expression $e(v_1, v_2, \ldots, v_k)$, $v_i \in \mathcal{V}$, $1 \leq i \leq k$, over the views in $\mathcal{V}$, which is equivalent to q. In other words, for any database $\mathcal{D}$, $e(v_1, v_2, \ldots, v_k)(\mathcal{D}) = q(\mathcal{D})$.*

In the above definition, *complete* rewritings are considered, that is: e only relies on the views and may not even have access to the database $\mathcal{D}$ any more. *Partial* rewritings have also been considered in the literature; a partial rewriting e' of a query q using the views in the set $\mathcal{V}$ is an expression $e'(v_1, v_2, \ldots, v_k, \mathcal{D})$ relying on the views *and* potentially on the database itself.

View-based query rewriting problem. *Given a query q and a set of views $\mathcal{V}$, the problem of* view-based query rewriting *consists of finding all the equivalent rewritings of q using the views in $\mathcal{V}$.*

The view-based query rewriting problem is declined in two variants, one considering only complete rewritings and another based on partial rewritings. A comprehensive survey on answering queries using views in relational databases is given in [Hal01]. Cost-based rewritings [DPT99, GL01, PH01] have been used to optimize query execution, while logical rewritings [YL87, LMSS95, AD98] are mostly used in the context of data integration from different sources.

View selection. *Given a query workload $\mathcal{Q}$ and a cost function c quantifying the cost of evaluating any rewriting r of a query $q \in \mathcal{Q}$ using a set of views $v_1, v_2, \ldots, v_k$, the problem of* view selection *consists of chosing a set of views $v_1^*, v_2^*, \ldots, v_n^*$ so that the overall cost of answering the workload queries, namely: $\Sigma_{q \in \mathcal{Q}} c(e(q, v_1^*, v_2^*, \ldots, v_n^*))$, is minimized.*

View selection has been extensively studied, especially in the context of data warehouses [HRU96, Gup97, TS97, ACN00]. Formal results on the complexity of the view selection problem are provided in [CHS02].

2.3 Cloud computing

The recent development of commercial cloud computing environments has strongly impacted research and development in distributed software platforms.

From a business perspective, cloud-based platforms release the application owner from the burden of administering the hardware, by providing elastic scaling up and down of resources according to the demand as well as reliable execution of tasks even in the face of machine failure.

From a research perspective, cloud platforms are interesting candidates to manage large data repositories. The new opportunities and potential performance issues arising in cloud-based data management platforms have been extensively discussed over the last few years [Aba09].

2.3.1 Cloud principles

The general idea behind *cloud computing* is not novel, dating back to the 1960s. Professor John McCarthy pronounced the following words at MIT's centennial celebration in 1963 [1], which characterized what we know today as *utility computing*:

> *"Computing may someday be organized as a public utility just as the telephone system is a public utility. Each subscriber needs to pay only for the capacity he actually uses, but he has access to all programming languages characteristic of a very large system ... Certain subscribers might offer service to other subscribers ... The computer utility could become the basis of a new and important industry."*

In the past, the form of utility computing that had enjoyed the biggest success was grid computing [FK99b]. In particular, grid computing was widely used in the research community over the last decade, e.g., to run large analysis processes on scientific data.

Cloud computing combines ideas from the grid computing paradigm, and other relevant technologies such as cluster computing, and distributed systems in general [FZRL09, AFG+10]. In particular, it provides off-site access, i.e., through the Internet, to shared resources in an *on demand* fashion. The main aspects that characterize cloud computing solutions are:

- *Scalability.* The ability to add more computing resources and being able to retain similar performance levels.
- *Elasticity.* The possibility to dynamically adapt to changes in the demand quickly, by provisioning and deprovisioning resources autonomously.
- *Pay as you go.* Users can request resources on demand and pay only for what they are using. This eliminates the need to invest up-front and to plan ahead.
- *Maintenance and fault tolerance.* The service provider is the responsible for taking care of the burden of managing the systems and dealing with possible failures, trying to avoid services outages.

The wide success of cloud computing is not technology-driven, but driven by economies of scale [FZRL09, AFG+10]. Cloud computing allows companies to outsource their IT infrastructure, and shifts the risk of provisioning to the cloud providers, which further decreases the cost.

2.3.2 Commercial clouds

Cloud services providers have released offers following various models. The services that they provide follow a *pay as you go* model that ensures that users pay for only what they are using.

One of the models is *Software as a Service (SaaS)*, which is based on the concept of renting software from a service provider rather than buying it yourself. The service provider hosts both the application and the data, so the end user does not have to

1. http://www.technologyreview.com/news/425623/the-cloud-imperative/

worry about anything and can use the service from anywhere. SaaS is very effective in lowering the costs of business as it provides an access to applications at a cost normally far cheaper than a licensed application fee.

Platform as a Service (PaaS) offers a development platform. The end users of PaaS write their own code and the provider executes that code in its infrastructure. PaaS provides services to develop, test, deploy, host and maintain applications in the same integrated development environment. Thus PaaS offers a more cost effective model for application development and delivery.

Infrastructure as a Service (IaaS) delivers the computing infrastructure as a fully outsourced service. The user can buy the infrastructure according to the requirements at any particular moment in an elastic manner instead of buying an infrastructure that might not be used for months. Virtualization enables IaaS providers to offer almost unlimited instances of servers to customers and make cost-effective use of the hosting hardware.

Several major IT companies have positioned themselves as providers of technology for these scenarios, e.g., Amazon Web Services [AWS], Google Cloud Compute [GCP], or Windows Azure [WA]. *Further information about the offer of each of these companies is given in Chapter 3, which builds our Web data repository using services they provide.*

2.3.3 Data repositories in commercial clouds

The proliferation of commercial cloud providers has led to an increasing interest in studying data storage and processing using the services that they offer.

The first work to propose a database architecture that exploited a commercial cloud, namely Amazon Web Services, was [BFG+08]. They build a relational database with a strong focus on protocols to provide certain transactional guarantees, e.g., atomicity or durability, while preserving the scalability and availability provided by the cloud. In turn, different alternative architectures for database applications in the cloud are studied in [KKL10], which evaluates the impact of the architectural choice on performance, scalability and cost of the application.

2.3.4 Pricing in the cloud

Recent works have focused on providing pertinent cost models for data management in the cloud. The novelty comes, on one hand, from the monetary costs associated to running a cloud application, and on the other hand, from the elasticity associated to managing data in the cloud.

The correlation between the consistency guarantees that a database running in the cloud can offer, and its associated cost, is studied in [KHAK09]. Further, different policies to choose among these consistency protocols are proposed.

Trade-offs between providing efficient query services and maintaining a profitable cloud are discussed in [DKA09]. They propose an economic model for self-tuned cloud caching of data, that encouraged high-quality service while reinforcing the profit. An adaptive optimal pricing schema is then proposed in [KDF+11].

The subject of building data access support structures (indexes and/or materialized views) to be shared by several applications running in the cloud is studied in [KDGA11]. In this context, the decision whether or not an index is useful (whether the costs associated to materializing the index are worth the improvement they bring to the multiple hosted applications) needs to be taken considering the history of a query mix and predicting the future use based on this history.

A framework that aim at scheduling data processing workflows that involve arbitrary data processing operators is presented in [KSTI11]. The schedule can be created attending to different goals, e.g. minimize completion time given a fixed budget or minimize monetary cost given a deadline.

2.3.5 Reliability, multi-tenancy and elasticity

Cloud-based data management raises several issues related to the distributed systems and reliability areas.

First of all, cloud-hosted data management is very often a multi-tenancy context, where several applications, potentially running for distinct organizations, share resources within the same distributed computation infrastructure. This implies that such applications have to compete for resources, but also that the global optimization and the pricing of resource consumption needs to take into account this sharing. The Kairos system [CJMB11] tries to consolidate databases onto a few physical machines and hence consume less energy. The process of consolidation involves analyzing the load characteristics of multiple dedicated database servers and packing their workloads into a few physical machines, reducing the resources consumed without changing the application performance.

Second, one of the core advantages of the cloud, namely, elastic allocation of resources to various applications, raises a set of issues on how to redistribute data across various cloud machines when scaling up and down an application. Also of importance, given the large sets of machines that may be involved in a given data management task, is coping with possible failures. The problem of migrating databases across several machines with minimal service interruption and no downtime, which is critical to allow lightweight elastic scaling, is studied in [EDAA11]. They propose a technique to migrate a database in a shared nothing transactional database architecture providing ACID guarantees during migration and/or failures.

2.4 Parallel processing systems

Besides parallel databases [DGG+86, FKT86], to scale complex data processing up to very large data volumes applications are increasingly relying on *implicit parallel* frameworks [DG04, BEH+10, ZCD+12] built on top of distributed filesystems. The main advantage of using such frameworks is that processing is distributed across many sites without the application having to explicitly handle data fragmentation, fragment placement etc.

2.4.1 Google File System and MapReduce

Google File System. The Google File System (GFS) [GGL03] was introduced by Google to meet their rapidly increasing data storage demands. GFS is a scalable distributed file system that provides fault tolerance guarantees while running on large shared-nothing clusters of commodity machines. GFS is optimized for workloads that consist mainly of (i) large streaming reads, and (ii) writes that append data to existing or new files, rather than overwriting existing data.

MapReduce. The original MapReduce [DG04] introduces a programming model and an associated implementation built on top of GFS. Inspired by the *map* and *reduce* functions commonly used in functional programming, MapReduce automatically parallelizes and executes programs over the files stored in GFS.

A MapReduce job consists mainly of two phases:

1. A *map* phase that takes key/value pairs as input and applies a user function on each of them, producing intermediary results in the form of key/value pairs.
2. A *reduce* phase that groups together all key/value pairs sharing the same key that were generated by the previous step, and invokes a user function once for each group. Between the map and the reduce phases, the data is *shuffled*, i.e., exchanged and merge-sorted.

Further, a reader function controls how the key/value pairs needed as input to the map phase are generated from the files in GFS, while a writer function describes how the output of the reduce phase is written to GFS.

As the MapReduce system takes care of tasks like partitioning the input data, scheduling the program's execution across a set of machines or handling machines failures, programmers are only requested to specify the user functions used in the map and reduce phases in order to utilize the resources of a large distributed system.

2.4.2 The Hadoop ecosystem

Soon after GFS and MapReduce were proposed by Google, Apache Hadoop [Had] was released. The Hadoop framework implemented open-source equivalents of the aforementioned systems, namely the Hadoop Distributed File System (HDFS) and Hadoop MapReduce. Its development was powered by contributors from enterprises such as Yahoo!, Facebook, Twitter, and other Web companies, which had a clear interest in using the platform to run a wide variety of data intensive tasks.

Since Hadoop was released, the number of systems built around it has not stop increasing, e.g., Apache Giraph [Gir] for graph data processing or Apache Storm [Sto] for real-time stream data processing. Recently, Apache Hadoop YARN [VMD+13] has been released. YARN decouples the resource management from the processing model, i.e., MapReduce, which had been identified as one of the shortcomings of Hadoop. This will enable a broader array of interaction patterns for data stored in HDFS, as the YARN-based architecture would not be anymore constrained to MapReduce. Surely, the emergence of YARN will only accelerate the proliferation of new proposals

built around the Hadoop ecosystem. In fact, new companies such as Cloudera [2] or Hortonworks [3] have already built their complete offer of data analytics services for enterprises around the Hadoop ecosystem.

2.4.3 Complex operations using MapReduce

Since MapReduce appeared, many works from industry and academia have focused on evaluating *efficiently* complex operations using the proposed model. The recents [LOOW14] and [DN14] provide detailed overviews on the state of the art on data processing using MapReduce.

2.4.4 Higher-level languages on top of MapReduce

As practitioners argued that MapReduce was too low-level and rigid to express some complex data processing tasks in a straightforward fashion, there was a proliferation of proposals for automatically translating higher-level languages (more declarative) into massively parallel frameworks such as MapReduce.

Pig Latin [ORS+08] is a dataflow language which allows specifying procedural style programs performing large-scale data processing tasks. Pig Latin is implemented into the Pig system [Piga] on top of Hadoop. After a Pig Latin script is parsed, a plan consisting of logical operators arranged in a DAG is generated and optimized. Then this plan is compiled in several stages into MapReduce. Pig Latin uses a nested data model, but it limits the levels of nested processing in its queries to guarantee acceptable parallelization of its computation. *Further information about Pig Latin is given in Chapter 5, which proposes a novel reuse-based optimizer for the language.*

Hive [TSJ+10] is a data warehousing solution built on top of Hadoop. Hive structures data into well-understood database concepts like tables, columns rows, and partitions. These structures are mapped into HDFS, and the corresponding metadata is kept in a metastore. Hive uses a nested relational data model, so complex types such as lists or maps can be embedded into values. Finally, queries are expressed in HiveQL, a subset of SQL extended with some features that are useful in their warehouse environment, and which is compiled into MapReduce.

Jaql [BEG+11] is a declarative scripting language tailored analytics of JSON data using Hadoop MapReduce. Jaql treats functions as first class citizens. For instance, Jaql expressions can be encapsulated in functions, and these functions can be parameterized later on at runtime. Moreover, Jaql's evaluation plan is entirely expressible in Jaql's syntax. This means that Jaql exposes every internal physical operator as a function in the language, allowing users to combine different levels of abstraction within a single script.

2. http://www.cloudera.com/
3. http://hortonworks.com/

2.4.5 XML data management on Hadoop

Recently, several works have studied XML storage and query processing techniques using Hadoop.

MRQL [FLGP11] proposes a simple SQL-like XML query language implemented through a few operators directly compilable into MapReduce. MRQL queries may be nested, and the XML navigation supported is XPath. In turn, ChuQL [KCS11] is an XQuery extension that exposes the MapReduce framework to the developer in order to distribute computations among XQuery engines.

Other recent works have proposed XML document partitioning techniques for HDFS. These approaches aim at querying efficiently a very large XML document using MapReduce. For instance, HadoopXML [CLK+12] statically partitions an input document into several blocs before storing it in HDFS, and some path information is added to blocs to avoid loss of structural information. The system assumes the query workload to be known in advance. The recent approach proposed in [BCM+13] dynamically fragments the input document by means of path information extracted from queries and updates.

2.4.6 Reuse-based optimizations on Hadoop

Multiple works have focused on avoiding redundant processing for a batch of MapReduce jobs by sharing their (intermediate) results.

[AKO08] studied the problem of scheduling jobs that need to access the same files simultaneously. They aim at maximizing the rate of processing these files by sharing scans of the same file as aggressively as possible, which is important for jobs whose execution time is dominated by data access.

MRShare [NPM+10] aims at identifying different jobs that share portions of identical work, so that those parts do not need to be recomputed from scratch. The main focus of MRShare is to maximize the total I/O savings, and therefore, sharing opportunities are identified in terms of sharing scans and sharing map output.

ReStore [EA12] is a system that manages the storage and reuse of intermediate results produced by workflows of MapReduce jobs. ReStore is implemented as an extension to the Pig system [Piga]. It is capable of reusing the output of whole MapReduce jobs, but it also creates additional reuse opportunities by materializing and storing the output of query execution operators that are executed within a MapReduce job. In order to do that, it maintains, together with a file storing a job's output, the physical execution plan of the query and some statistics about the job that produced it, as well as how frequently this output is used by other workflows.

Finally, the recent [WC13] proposes two new techniques for the optimization of a batch of MapReduce jobs: (i) a generalization of MRShare that allows further sharing of map outputs, and (ii) a materialization and reutilization technique that relies on choosing an appropiate execution order of jobs in the input batch. Further, it presents an algorithm to produce an optimal plan for a given batch of jobs, by partitioning them into groups and assignment a different processing technique to each group.

2.4.7 Other parallel processing stacks

Following the success of the original MapReduce proposal, other systems and models have been proposed, which try to extend the MapReduce paradigm by eliminating some shortcomings of the original model and extending its expressive power.

Stratosphere [4] [ABE+14, Str] proposes a new stack on top of HDFS. In particular, the key components of the system are:

- A new programming model, the *PArallelization ConTracts* [BEH+10] (PACT, in short). PACT extends MapReduce by (*i*) manipulating records with any number of fields, instead of key/value pairs, (*ii*) proposing a richer set of operators based on second-order functions, and (*iii*) allowing one parallel operator to receive as input the outputs of several other such operators.
- Its own high performance execution engine, Nephele [WK09]. A cost-based optimizer compiles PACT programs down to dataflow graphs that can be massively parallelized by the Nephele engine.

Stratosphere is a very active project, and recent works have proposed an optimizer for PACT plans [HPS+12], or have studied how to efficiently integrate the execution of iterative processing tasks into the platform [ETKM12, SETM13]. Further, [HRL+12] presents Meteor, a scripting language heavily inspired by Jaql [BEG+11] that compiles into PACT.

Further information about the Stratosphere system is given in Chapter 4, as we build our massively parallel XQuery processor on top of the PACT model.

Apache Spark [ZCF+10, Spa] is another large-scale data processing engine that can run on top of HDFS. The main abstraction that Spark provides is the *Resilient Distributed Dataset* [ZCD+12] (RDD, in short), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. Two types of operations can be applied on an RDD: a *transformation*, which creates a new dataset from an existing one, and an *action*, which returns a value to the driver program after running a computation on the dataset. Further, users may ask Spark to persist an RDD in memory, allowing it to be reused efficiently across parallel operations, which makes the model specially well suited for executing bulk iterative algorithms. Finally, RDDs can automatically recover from node failures.

AsterixDB [BBC+11, Ast] is another effort that proposes a new parallel semistructured information management system, and whose architectural changes are more radical with respect to the original MapReduce and the approaches presented above. AsterixDB consists of three main components: (*i*) the Hyracks execution engine [BCG+11], (*ii*) the Algebricks algebra, which compiles into parallel plans executable by Hyracks, and (*iii*) the Asterix Query Language, or AQL in short, based on *FLWOR* expressions, that compiles into Algebricks. Although Algebricks is the target for AQL, it is currently used as the compilation target for other declarative query languages e.g., HiveQL or Pig Latin. Moreover, instead of working on top of HDFS,

4. Stratosphere has been recently accepted as an Apache incubator project and will be renamed to Apache Flink (http://incubator.apache.org/projects/flink.html).

AsterixDB proposes its own native storage manager with support for indexes, etc.

The VXQuery project [VXQ] is an ongoing effort that aims at parallelizing XQuery in the AsterixDB system by translating XQuery into the Algebricks algebra.

2.5 Conclusion

In this chapter we presented XML, a popular data model for representing and sharing data on the Web. Further, we discussed different optimization techniques whose foundations are in the query equivalence and containment notions. Then, we introduced briefly the ideas behind the *cloud* computing concept, and the research challenges it raises. Finally, we presented different frameworks that rely heavily on parallelization to scale processing up to large data volumes.

Chapter 3

AMADA: Web Data Repositories in the Cloud

In this chapter, we present AMADA, an architecture for warehousing large-scale Web data in commercial cloud platforms. AMADA operates in a Software as a Service (SaaS) approach, allowing users to upload, store, and query large volumes of Web data. Since cloud users support monetary costs directly connected to their consumption of resources, our focus in this work is not only on query performance from an execution time perspective, but also on the monetary costs associated to this processing. In particular, we study the applicability of several content indexing strategies, and show that they lead not only to reducing query evaluation time, but also, importantly, to reducing the monetary costs associated with the exploitation of the cloud-based warehouse.

An early version of this work was presented in a workshop [CRCM12], while this chapter closely follows the international conference publication [CRCM13]. RDF data management using the same architecture that we present here was studied in parallel in the team, including as part of this thesis work [BCRG+14]. The AMADA system was demonstrated in [AABCR+12], and finally open-sourced in March 2013 [1].

3.1 Introduction

Over the last few years, big IT companies such as Amazon, Google or Microsoft have started providing an increasing number of cloud services built on top of their infrastructure. Using these commercial cloud platforms, organizations and individuals can take advantage of a deployed infrastructure and build their applications on top of it. An important feature of such platforms is their elasticity, i.e., the ability to allocate more (or less) computing power, storage, or other services, as the application demands grow or shrink. Cloud services are rented out based on specific service-level agreements (SLAs) characterizing their performance, reliability etc.

1. http://cloak.saclay.inria.fr/research/amada/

Although the services offered by public clouds vary, they all provide some form of scalable, durable, highly-available store for files, or (equivalently) binary large objects. Cloud platforms also provide virtual machines (typically called *instances*) which are started and shut down as needed, and on which one can actually deploy code to be run. This gives a basic roadmap for warehousing large volumes of Web data in the cloud in a Software as a Service (SaaS) mode:

- To *store the data*, load it in the cloud-based file store.
- To *process a query*, deploy some instances, have them read data from the file store, compute query results and return them.

Clearly, the performance (response time) incurred by this processing is of importance; however, so are the monetary costs associated to this scenario, that is, the costs to load, store, and process the data for query answering. The costs billed by the cloud provider, in turn, are related to the total effort (or total work), in other terms, the total consumption of all the cloud resources, entailed by storage and query processing. In particular, when the warehouse is large, if query evaluation involves all (or a large share of) the data, this leads to high costs for: (i) reading the data from the file store and (ii) processing the query on the data.

In this chapter, we investigate the usage of *content indexing*, as a tool to both improve query performance, and reduce the total costs of exploiting a warehouse of Web data within the cloud.

We focus on tree-structured data, and in particular XML, due to the large adoption of this and other tree-shaped formats such as JSON, and we consider the particular example of the Amazon Web Services (AWS, in short) platform, among the most widely adopted, and also target of previous research works [BFG$^+$08, SDQR10]. Since there is a strong similarity among commercial cloud platforms, our results could easily carry on to another platform (we briefly discuss this in Section 3.2). The contributions of the chapter are the following:

- A generic architecture for large-scale warehousing of complex structured data (in particular, tree-shaped data) in the cloud. Our focus is on building and exploiting various kinds of indexing strategies to simultaneously speed up processing and reduce cloud resource consumption (and thus, warehouse operating costs);
- A concrete implemented platform following this architecture, demonstrating its practical interest and validating the claimed benefits of our indexes through experiments on a 40 GB dataset. We show that indexing can reduce processing time by up to two orders of magnitude and costs by one order of magnitude; moreover, index creation costs amortize very quickly as more queries are run.

The remainder of this chapter is organized as follows. Section 3.2 describes AMADA's architecture, while Section 3.3 introduces its associated monetary cost model. Section 3.4 presents our query language, while Section 3.5 focuses on cloud-based indexing strategies. Section 3.6 details our system implementation on top of AWS, while Section 3.7 provides our experimental evaluation results. Section 3.9 discusses related works. We then conclude and outline future directions.

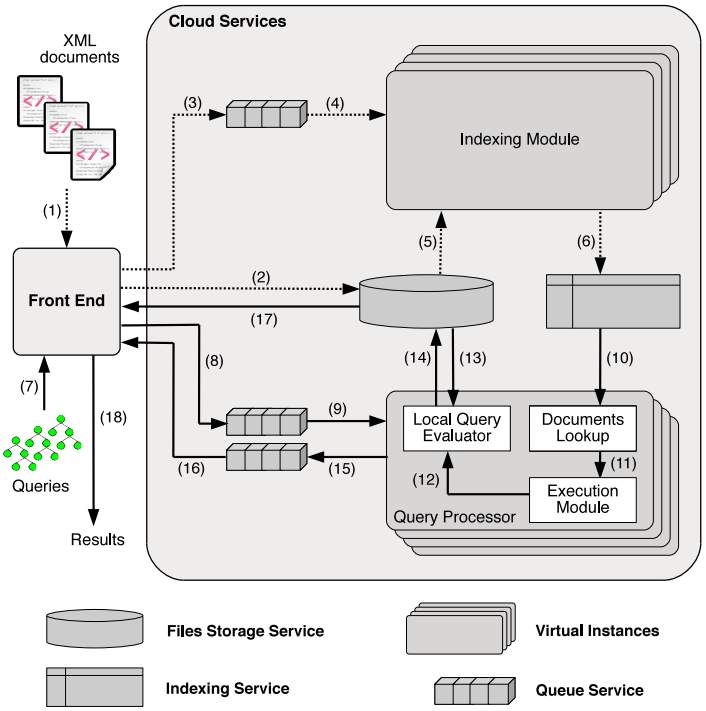


Figure 3.1: Architecture overview.

3.2 Architecture

We now describe AMADA, our proposed architecture for Web data warehousing. To build a scalable, inexpensive data store, we store documents as files within Amazon's Simple Storage Service (S3, in short). To host the index, we have different requirements: fine-grained access, and fast look-up. Thus, we rely on Amazon's DynamoDB efficient key-value store for storing and exploiting the index. Within AWS, instances can be deployed through the Elastic Compute Cloud service (EC2, in short). We deploy EC2 instances to (i) extract from the loaded data index entries, and send them to Dynamo DB's index and (ii) evaluate queries on those subsets of the database resulting from the query-driven index lookups. Finally, we rely on Amazon Simple Queue Service (SQS, in short) asynchronous queues to provide reliable communication between the different modules of the application. Figure 3.1 depicts this architecture.

User interaction with the system involves the following steps.

– When a document arrives (1), the front end of our application stores it in the file storage service (2). Then, the front end module creates a message containing the reference to the document and inserts it into the *loader request* queue (3). Such messages are retrieved by any of the virtual machines running our indexing module (4). When a message is retrieved, the application loads the document referenced by the message from the file store (5) and creates the index data that is finally inserted into the index store (6).

– When a query arrives (7), a message containing the query is created and in-

Provider	**File store**	**Key-value store**	**Computing**	**Queues**
Amazon Web Services [AWS]	Amazon Simple Storage Service	Amazon DynamoDB, Amazon SimpleDB	Amazon Elastic Compute Cloud	Amazon Simple Queue Service
Google Cloud Platform [GCP]	Google Cloud Storage	Google High Replication Datastore	Google Compute Engine	Google Task Queues
Windows Azure [WA]	Windows Azure BLOB Storage	Windows Azure Tables	Windows Azure Virtual Machines	Windows Azure Queues

Table 3.1: Component services from major commercial cloud platforms.

serted into the *query request* queue (8). Such messages are retrieved by any of the virtual instances running our query processor module (9). The index data is then extracted from the index store (10). Index storage services provide an API with rather simple functionalities typical of key-value stores, such as *get* and *put* requests. Thus, any other processing steps needed on the data retrieved from the index are performed by a standard XML querying engine (11), providing value- and structural joins, selections, projections etc.
After the document references have been extracted from the index, the local query evaluator receives this information (12) and the XML documents cited are retrieved from the file store (13). Our framework includes "standard" XML query evaluation, i.e. the capability of evaluating a given query q over a given document d. This is done by means of a single-site XML processor, which one can choose freely. Thus, the virtual instance runs the query processor over the documents and extracts the results for the query.
Finally, we write the results obtained in the file store (14) and we create a message with the reference to those results and insert it into the *query response* queue (15). When the message is read by the front end (16), the results are retrieved from the file store (17) and returned (18).

Scalability, parallelism and fault-tolerance. The architecture described above exploits the elastic scaling of the cloud, for instance increasing and decreasing the number of virtual machines running each module. The synchronization through the message queues among modules provides inter-machine parallelism, whereas intra-machine parallelism is supported by multi-threading our code. We have also taken advantage of the primitives provided by message queues, in order to make our code resilient to a possible virtual instance crash: if an instance fails to renew its lease on the message which had caused a task to start, the message becomes available again and another virtual instance will take over the job.

Applicability to other cloud platforms. While we have instantiated the above architecture based on AWS, it can be easily ported to other well-known commercial clouds. Table 3.1 shows the services available in the Google and Microsoft cloud platforms, corresponding to the ones we use from AWS. It can be easily seen that these platforms are largely similar in terms of their provided software suites.

3.3 Application costs

A relevant question in a cloud-hosted application context is, how much will this cost? Commercial cloud providers charge specific costs for the usage of their services. This section presents our model for estimating the monetary costs of uploading, indexing, hosting and querying Web data in a commercial cloud. Section 3.3.1 presents the metrics used for calculating these costs, while Section 3.3.2 introduces the components of the cloud provider's pricing model relevant to our application. Finally, Section 3.3.3 proposes the cost model for storing and indexing the data, and for answering queries.

3.3.1 Data set metrics

The following metrics capture *the impact of a given dataset, indexing strategy and query* on the charged costs.

Data-dependent metrics. Given a set of documents $\mathscr{D}$, $|\mathscr{D}|$ and $s(\mathscr{D})$ indicate the number of documents in $\mathscr{D}$, and the total size (in GB) of all the documents in $\mathscr{D}$, respectively.

Data- and index-determined metrics. For a given set of documents $\mathscr{D}$ and indexing strategy I, let $|put(\mathscr{D}, I)|$ be the number of *put* requests needed to store the index for $\mathscr{D}$ based on strategy I.

Let $t(\mathscr{D}, I)$ be the time needed to build and store the index for $\mathscr{D}$ based on strategy I. The meaningful way to measure this in our framework (recall the processing stages outlined in Section 3.2) is: *the time elapsed between*

- *the moment when the first message (document loading request) entailed by loading $\mathscr{D}$ is retrieved from our application's queue, until*
- *the moment when the last such message was deleted from the queue*.

To compute the index size, we use:

- $sr(\mathscr{D}, I)$ is the raw size (in GB) of the data extracted from $\mathscr{D}$ according to I.
- Cloud key-value stores (in our case, DynamoDB) create their own auxiliary data structures, on top of the user data they store. We denote by $ovh(\mathscr{D}, I)$ the size (in GB) of the storage overhead incurred for the index data extracted from $\mathscr{D}$ according to I.

Thus, the size of the data stored in an indexing service is:

$$s(\mathscr{D}, I) = sr(\mathscr{D}, I) + ovh(\mathscr{D}, I)$$

Data-, index- and query-determined metrics. First, let $s(q, \mathscr{D})$ be the size (in GB) of the results for query q over the documents set $\mathscr{D}$.

When using the indexing strategy I, let $|get(q, \mathscr{D}, I)|$ be the number of *get* operations needed to look up documents that may contain answers to q, in an index for $\mathscr{D}$ built based on the strategy I. Similarly, let $\mathscr{D}_I^q$ be the subset of $\mathscr{D}$ resulting from look-up on the index built based on strategy I for query q.

Let $t(q, \mathscr{D})$ be the time needed by the query processor to answer a query q over a dataset D without using any index, and $t(q, \mathscr{D}, I, \mathscr{D}_I^q)$ be the time to process q with the support of an index built according to the strategy I (thus, on the reduced document set $\mathscr{D}_I^q$), respectively. We measure it as the time elapsed from the moment the message with the query was retrieved from the queue service to the moment the message was deleted from it.

3.3.2 Cloud services costs

We list here the costs spelled out in the cloud service provider's pricing policy, which impact the costs charged by the provider for our Web data management application.

File storage costs. We consider the following three components for calculating costs associated to a file store.

- $ST^{\$}_{m,GB}$ is the cost charged for storing and keeping 1 GB of data in a file store for one month.
- $ST^{\$}_{put}$ is the price per document storage operation request.
- $ST^{\$}_{get}$ is the price per document retrieval operation request.

Indexing costs. We consider the following components for calculating the index store associated costs.

- $IDX^{\$}_{m,GB}$ is the cost charged for storing and keeping 1 GB of data in the index store for one month.
- $IDX^{\$}_{put}$ is the cost of a *put* API request that inserts a row into the index store.
- $IDX^{\$}_{get}$ is the cost of a *get* API request that retrieves a row from the index store.

Virtual instance costs. $VM^{\$}_h$ is the price charged for running for one hour a virtual machine. As cloud providers offer different types of instances with different hardware specifications, the price depends on the kind of virtual machine we choose to use.

Queue service costs. $QS^{\$}$ is the price charged for each request to the queue service API, e.g., send message, receive message, delete message, renew lease etc.

Data transfer. The commercial cloud providers considered in this work do not charge anything for data transferred to or within their cloud infrastructure. However, data transferred out of the cloud incurs a cost: $egress^{\$}_{GB}$ is the price charged for transferring 1 GB out of the cloud.

3.3.3 Web data management costs

We now show how to compute, based on the data-, index- and query-driven metrics (Section 3.3.1), together with the cloud service costs (Section 3.3.2), the costs incurred by our Web data storage architecture in a commercial cloud. Table 3.2 provides an overview of the notations introduced throughout Section 3.3.1 and Section 3.3.2.

Notation	Description
$\mathscr{D}$	Documents set.
I	Indexing strategy.
q	Query.
$\mathscr{D}_I^q$	Subset of $\mathscr{D}$ resulting from look-up for q on the index built based on I.
$s(\mathscr{D})$	Total size of the documents in $\mathscr{D}$.
$s(\mathscr{D}, I)$	Total size of the index built for $\mathscr{D}$ based on strategy I.
$s(q, \mathscr{D})$	Total size of the results for q over the documents in $\mathscr{D}$.
$\|\mathscr{D}\|$	Number of documents in $\mathscr{D}$.
$\|put(\mathscr{D}, I)\|$	Number of *put* operations to store the index for $\mathscr{D}$ based on strategy I.
$\|get(q, \mathscr{D}, I)\|$	Number of *get* operations to look up documents that may contain answers to q in an index for $\mathscr{D}$ built based on the strategy I.
$t(\mathscr{D}, I)$	Time to build and store the index for $\mathscr{D}$ based on strategy I.
$t(q, \mathscr{D})$	Time to answer q over D without using any index
$t(q, \mathscr{D}, I, \mathscr{D}_I^q)$	Time to answer q over D using an index built according to the strategy I that returns a reduced document set $\mathscr{D}_I^q$.
$ST^{\$}_{m,GB}$	Price charged for storing 1 GB of data for one month (file store).
$ST^{\$}_{put}$	Price per storage operation request (file store).
$ST^{\$}_{get}$	Price per retrieval operation request (file store).
$IDX^{\$}_{m,GB}$	Price charged for storing 1 GB of data for one month (index store).
$IDX^{\$}_{put}$	Price per storage operation request (index store).
$IDX^{\$}_{get}$	Price per retrieval operation request (index store).
$VM^{\$}_{h}$	Price charged for running a virtual machine for one hour.
$QS^{\$}$	Price per operation request to the queue service.
$egress^{\$}_{GB}$	Price charged for transferring 1 GB out of the cloud.

Table 3.2: Data set metrics and cloud services costs associated notations.

Storing and indexing the data. Given a set of documents $\mathscr{D}$, we calculate the cost of uploading it into a file store as follows:

$$ud^{\$}(\mathscr{D}) = ST^{\$}_{put} \times |\mathscr{D}| + QS^{\$} \times |\mathscr{D}|$$

Thus, the cost of building the index for $\mathscr{D}$ by means of the indexing strategy I is:

$$\begin{aligned} ci^{\$}(\mathscr{D}, I) \quad = \quad & ud^{\$}(\mathscr{D}) + ST^{\$}_{get} \times |\mathscr{D}| + IDX^{\$}_{put} \times |put(\mathscr{D}, I)| \\ & + VM^{\$}_{h} \times t(\mathscr{D}, I) + QS^{\$} \times 2 \times |\mathscr{D}| \end{aligned}$$

Note that we need two queue service requests for each document: the first obtains the URI of the document that needs to be processed, while the second deletes the message from the queue when the document has been indexed.

The cost of storing $\mathscr{D}$ in the file store and the index structure created for $\mathscr{D}$ according to I in the index store for one month is calculated as:

$$st^{\$}_{m}(\mathscr{D}, I) = ST^{\$}_{m,GB} \times s(\mathscr{D}) + IDX^{\$}_{m,GB} \times s(\mathscr{D}, I)$$

Querying. First, we estimate the cost incurred by the front-end for sending query q and retrieving its results as:

$$rq^{\$}(q, \mathscr{D}) = ST^{\$}_{get} + egress^{\$}_{GB} \times s(q, \mathscr{D}) + QS^{\$} \times 3$$

Three queue service requests are issued: the first one sends the query, the second one retrieves the reference to the query results, and the third one deletes the message retrieved by the second request.

The cost for answering a query q without using any index is calculated as follows:

$$\begin{aligned} cq^{\$}(q, \mathscr{D}) \quad = \quad & rq^{\$}(q, \mathscr{D}) + ST^{\$}_{get} \times |\mathscr{D}| + ST^{\$}_{put} \\ & + VM^{\$}_{h} \times t(q, \mathscr{D}) + QS^{\$} \times 3 \end{aligned}$$

Note that, again, three queue service requests are issued: the first one retrieves the message containing the query, the second one sends the message with the reference to the results for the query, while the third removes the message with the query from the corresponding queue. The same holds for the formula below which calculates the cost of evaluating a query q over $\mathscr{D}$ indexed according to I:

$$\begin{aligned} cq^{\$}(q, \mathscr{D}, I, \mathscr{D}^q_I) \quad = \quad & rq^{\$}(q, \mathscr{D}) + IDX^{\$}_{get} \times |get(q, \mathscr{D}, I)| + ST^{\$}_{get} \times |\mathscr{D}^q_I| + ST^{\$}_{put} \\ & + VM^{\$}_{h} \times t(q, \mathscr{D}, I, \mathscr{D}^q_I) + QS^{\$} \times 3 \end{aligned}$$

This finalizes our monetary cost model for Web data stores in commercial clouds, according to the architecture we described in Section 3.2. Some parameters of the cost model are well-known (those determined by the input data size and the provider's cost policy), while others are query- and strategy-dependent (e.g., how may documents match a query etc.) In Section 3.7 we measure actual charged costs, where the query- and strategy-dependent parameters are instantiated to concrete operations. These measures allow to highlight the cost savings brought by our indexing.

3.4 Query language

We consider queries are formulated in an expressive fragment of XQuery, amounting to value joins over tree patterns. For illustration, Figure 3.2 depicts some queries in a graphical notation which is easier to read. The translation from this tree pattern language to XQuery syntax, as well as the formal pattern semantics, can be found in [MKVZ11].

In a tree pattern, each node is labeled either with an XML element or attribute name. By convention, attribute names are prefixed with the `@` sign. Parent-child relationships are represented by single lines while ancestor-descendant relationships are encoded by double lines.

Each node corresponding to an XML element may be possibly annotated with (i) the label *val* if the string value of the element (obtained by concatenating all its text descendants) is needed, and/or (ii) the label *cont* if the full XML subtree rooted at

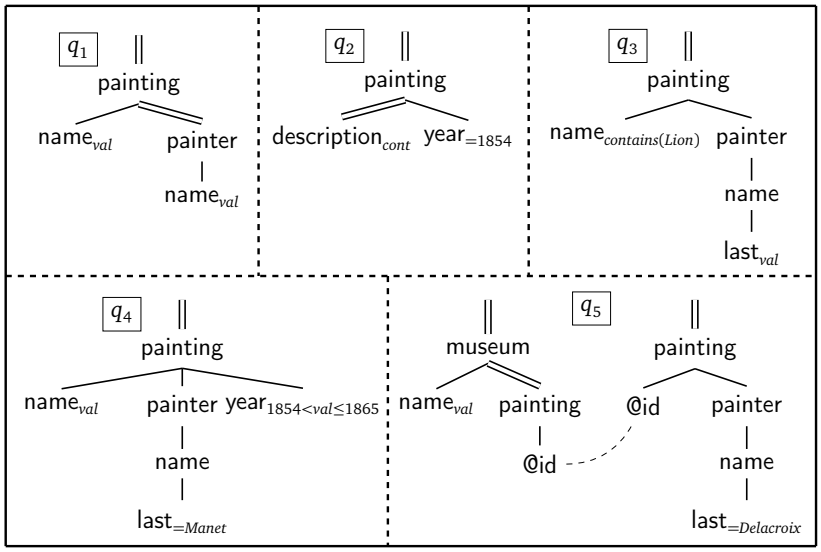


Figure 3.2: Sample queries.

this node is needed. We support these different levels of information for the following reasons. The *val*ue of a node is used in the XQuery specification to determine whether two nodes are equal (e.g., whether the name of a painter is the same as the name of a book writer). The *cont*ent is the natural granularity of XML query results (full XML subtrees are returned by the evaluation of an XPath query). A tree pattern node corresponding to an XML attribute may be annotated with the label *val*, denoting that the string value of the attribute is returned. Further, any node may also be annotated with one among the following predicates on its *val*ue:

- An equality predicate of the form $= c$, where c is some constant, imposing that the node *val*ue is equal to c.
- A containment predicate of the form *contains*(c), imposing that the node *val*ue contains the word c.
- A range predicate of the form $[a \leq val \leq b]$, where a and b are constants such that $a < b$, imposing that its *val*ue is inside that range.

Finally, a dashed line connecting two nodes joins two tree patterns, on the condition that the *val*ue of the respective nodes is the same on both sides.

For instance, in Figure 3.2, q_1 returns the pair (painting name, painter name) for each painting, q_2 returns the descriptions of paintings from *1854*, while q_3 returns the last name of painters having authored a painting whose name includes the word *Lion*. Query q_4 returns the name of the painting(s) by *Manet* created between *1854* and *1865*, and finally query q_5 returns the name of the museum(s) exposing paintings by *Delacroix*.

3.5 Indexing strategies

Many indexing schemes for semistructured data, and in particular XML, have been devised in the literature, e.g., [GW97, KBNK02, MS99]. In this Section, we explain

Name	Indexing function
LU	$I_{LU}(d) = \{(key(n),\ (URI(d), \epsilon)) \mid n \in d\}$
LUP	$I_{LUP}(d) = \{(key(n),\ (URI(d), \{inPath_1(n), inPath_2(n), \ldots, inPath_y(n)\})) \mid n \in d\}$
LUI	$I_{LUI}(d) = \{(key(n),\ (URI(d), id_1(n) \| id_2(n) \| \ldots \| id_z(n))) \mid n \in d\}$
2LUPI	$I_{2LUPI}(d) = \{[(key(n),\ (URI(d), \{inPath_1(n), inPath_2(n), \ldots, inPath_y(n)\})),$ $(key(n),\ (URI(d), id_1(n) \| id_2(n) \| \ldots \| id_z(n)))] \mid n \in d\}$

Table 3.3: Indexing strategies.

how we adapted a set of relatively simple XML indexing strategies, previously used in other distributed environments [AMP+08, GWJD03] into our setting, where the index is built within a heterogeneous key-value store.

Notations. Before describing the different indexing strategies, we introduce some useful notations.

In the following, d stands for an XML document, while q is a query expressed in the language described in Section 3.4. We denote by $URI(d)$ the Uniform Resource Identifier (or URI, in short) of d. For a given node $n \in d$, we denote by $inPath(n)$ the label path going from the root of d to the node n, and by $id(n)$ the node identifier (or ID, in short). We rely on simple (*pre*, *post*, *depth*) identifiers used, e.g., in [AKJP+02] and many follow-up works. Such IDs allow identifying if node n_1 is an ancestor of node n_2 by testing whether $n_1.pre < n_2.pre$ and $n_1.post < n_2.post$. If this is the case, n_1 is the parent of n_2 iff $n_1.depth + 1 = n_2.depth$.

For a given node $n \in d$, the function $key(n)$ computes a string key based on which n's information is indexed. Let $\underline{e}$, $\underline{a}$ and $\underline{w}$ be three constant string tokens, and $\|$ denote string concatenation. We define $key(n)$ as:

$$key(n) = \begin{cases} \underline{e}\|n.label & \text{if } n \text{ is an XML element} \\ \underline{a}\|n.name & \\ \underline{a}\|n.name\|n.val & \text{if } n \text{ is an XML attribute} \\ \underline{w}\|n.val & \text{if } n \text{ is a word} \end{cases}$$

Observe that we extract *two* keys from an attribute node, one to reflect the attribute name and another to reflect its value; these help speed up specific kinds of queries, as we will see. To simplify, we omit the $\|$ when this does not lead to confusion.

Indexing strategies. An *indexing strategy* I is a function which, from a set of documents, constructs a set of tuples of the form $(k, (a, v^+)^+)^+$. In other words, $I(d)$ represents the set of keys k that must be added to the index store to reflect the new document d, as well as the attributes to be stored on the respective key. Each attribute contains a name a and a set of values v.

Table 3.3 depicts the proposed indexing strategies. We explain them in detail in the following sections.

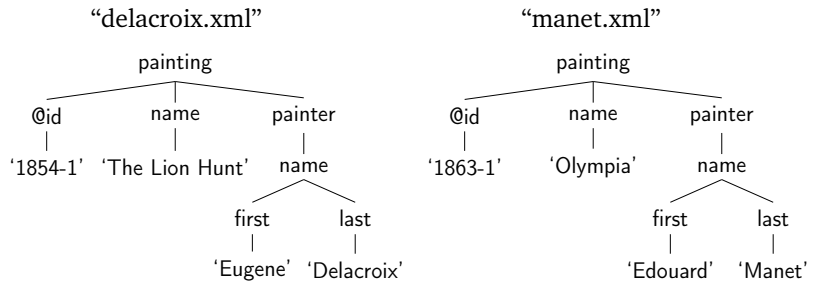


Figure 3.3: Sample XML documents.

3.5.1 Strategy *LU* (Label-URI)

Index. For each node $n \in d$, strategy *LU* associates to the key $key(n)$ the pair $(URI(d), \epsilon)$ where ϵ denotes the null string. For example, applied to the documents depicted in Figure 3.3, *LU* produces among others these tuples:

key	attribute name	attribute values
$\underline{e}$name	"delacroix.xml"	ϵ
	"manet.xml"	ϵ
$\underline{a}$id	"delacroix.xml"	ϵ
	"manet.xml"	ϵ
$\underline{a}$id 1863-1	"manet.xml"	ϵ
$\underline{w}$Olympia	"manet.xml"	ϵ

Look-up. The look-up task consists of finding, by exploiting the index, and as precisely as possible, those parts of the document warehouse that may lead to query answers.

Index look-up based on the *LU* strategy is quite simple: all node names, attribute and element string values are extracted from the query and the respective look-ups are performed. Then URI sets thus obtained are intersected. The query is evaluated on those documents whose URIs are part of the intersection.

3.5.2 Strategy *LUP* (Label-URI-Path)

Index. For each node $n \in d$, *LUP* associates to $key(n)$ the attribute:

$$(URI(d), \{inPath_1(n), \ldots, inPath_y(n)\})$$

On the "delacroix.xml" and "manet.xml" documents shown in Figure 3.3, extracted tuples include:

key	attribute name	attribute values
$\underline{e}$name	"delacroix.xml"	/$\underline{e}$painting/$\underline{e}$name, /$\underline{e}$painting/$\underline{e}$painter/$\underline{e}$name
	"manet.xml"	/$\underline{e}$painting/$\underline{e}$name, /$\underline{e}$painting/$\underline{e}$painter/$\underline{e}$name
$\underline{a}$id	"delacroix.xml"	/$\underline{e}$painting/$\underline{a}$id
	"manet.xml"	/$\underline{e}$painting/$\underline{a}$id
$\underline{a}$id 1863-1	"manet.xml"	/$\underline{e}$painting/$\underline{a}$id 1863-1
$\underline{w}$Olympia	"manet.xml"	/$\underline{e}$painting/$\underline{e}$name/$\underline{w}$Olympia

Look-up. The look-up strategy consists of finding, for each root-to-leaf path appearing in the query q, all documents having a data path that matches the query path. Here, a root-to-leaf query path is obtained simply by traversing the query tree and recording node keys and edge types. For instance, for the query q_1 in Figure 3.2, the paths are: //$\underline{e}$painting/$\underline{e}$name and //$\underline{e}$painting//$\underline{e}$painter/$\underline{e}$name.

To find the URIs of all documents matching a given query path

$$(/|//)a_1(/|//)a_2 \ldots (/|//)a_j$$

we look up in the *LUP* index all paths associated to $key(a_j)$, and then filter them to only those matching the path.

3.5.3 Strategy *LUI* (Label-URI-ID)

Index. The idea of this strategy is to concatenate the structural identifiers of a given node in a document, already *sorted* by their *pre* component, and store them into a single attribute value. We propose this implementation because structural XML joins which are used to identify the relevant documents need sorted inputs: thus, by keeping the identifiers ordered, we reduce the use of expensive sort operators after the look-up.

To each key $key(n)$, strategy *LUI* associates the pair

$$(URI(d), id_1(n)||id_2(n)|| \ldots ||id_z(n))$$

such that $id_1(n)<id_2(n)<\ldots<id_z(n)$. For instance, from the documents "delacroix.xml" and "manet.xml", some extracted tuples are:

key	attribute name	attribute values
$\underline{e}$name	"delacroix.xml"	(3, 3, 2)(6, 8, 3)
	"manet.xml"	(3, 3, 2)(6, 8, 3)
$\underline{a}$id	"delacroix.xml"	(2, 1, 2)
	"manet.xml"	(2, 1, 2)
$\underline{a}$id 1863-1	"manet.xml"	(2, 1, 2)
$\underline{w}$Olympia	"manet.xml"	(4, 2, 3)

Look-up. Index look-up based on *LUI* starts by searching the index for all the query labels. For instance, for the query q_2 in Figure 3.2, the look-ups will be $\underline{e}$painting, $\underline{e}$description, $\underline{e}$year and $\underline{w}$1854.

Then, in order to compute the query results by applying a Holistic Twig Join [BKS02] over the resulting node identifiers, one only needs to sort them in the increasing order of their URI (recall that the structural identifiers for any given document are already sorted).

1st key	1st attribute name	1st attribute values
ename	"delacroix.xml"	/epainting/ename, /epainting/epainter/ename
	"manet.xml"	/epainting/ename, /epainting/epainter/ename
aid 1863-1	"manet.xml"	/epainting/aid 1863-1
wOlympia	"manet.xml"	/epainting/ename/wOlympia

2nd key	2nd attribute name	2nd attribute values
ename	"delacroix.xml"	(3,3,2)(6,8,3)
	"manet.xml"	(3,3,2)(6,8,3)
aid	"delacroix.xml"	(2,1,2)
	"manet.xml"	(2,1,2)
aid 1863-1	"manet.xml"	(2,1,2)
wOlympia	"manet.xml"	(4,2,3)

Figure 3.4: Sample tuples extracted by the *2LUPI* strategy from the documents in Figure 3.3.

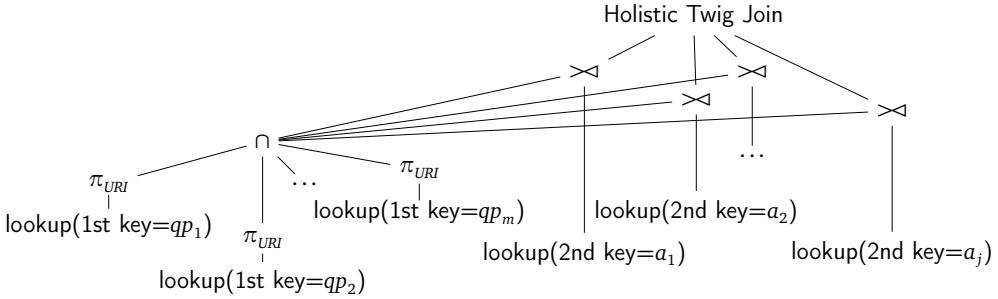


Figure 3.5: Outline of look-up using *2LUPI* strategy.

3.5.4 Strategy *2LUPI* (Label-URI-Path, Label-URI-ID)

Index. This strategy materializes two previously introduced indexes: *LUP* and *LUI*. Sample index tuples resulting from the documents in Figure 3.3 are shown in Figure 3.4.

Look-up. *2LUPI* exploits, first, *LUP* to obtain the set of documents containing matches for the query paths, and second, *LUI* to retrieve the IDs of the relevant nodes.

For instance, given the query q_2 in Figure 3.2, we extract the URIs of the documents matching //epainting//edescription and //epainting/eyear/w1854. The URI sets are intersected, and we obtain a relation which we denote $R_1(URI)$. This is reminiscent of the *LUP* look-up. A second look-up identifies the structural identifiers of the XML nodes whose labels appear in the query, together with the URIs on their documents. This reminds us of the *LUI* look-up. We denote these relations by $R_2^{a_1}$, $R_2^{a_2}$, ... $R_2^{a_j}$, assuming the query node labels and values are a_1, a_2, ..., a_j. Then:

- We compute $S_2^{a_i} = R_2^{a_i} \ltimes_{URI} R_1(URI)$ for each a_i, $1 \leq i \leq j$. In other words, we use $R_1(URI)$ to reduce the R_2 relations in the spirit of classical semijoin reducers [OV11].
- We evaluate a holistic twig join [BKS02] over $S_2^{a_1}$, $S_2^{a_2}$, ..., $S_2^{a_j}$ to obtain URIs of

the documents with query matches. The tuples for each input of the holistic join are obtained by sorting the attributes by their name and then breaking down their values into individual IDs.

Figure 3.5 outlines this process; $a_1, a_2, \ldots, a_j$ are the labels extracted from the query, while $qp_1, qp_2, \ldots, qp_m$ are the root-to-leaf paths extracted from the query.

It follows from the above explanation that *2LUPI* returns the same URIs as *LUI*. The reduction phase serves for prefiltering, to improve performance.

3.5.5 Range and value-joined queries

This type of queries which are supported by our language need special evaluation strategies.

Queries with range predicates. Range look-ups in key-value stores usually imply a full scan, which is very expensive. Thus, we adopt a two-step strategy. First, we perform the index look-up without taking into account the range predicate, in order to restrict the set of documents to be queried. Second, we evaluate the complete query over these documents, as usual.

Queries with value joins. Since one tree pattern only matches one XML document, a query consisting of several tree patterns connected by a value join needs to be answered by combining tree pattern query results from different documents. Indeed, this is our evaluation strategy for such queries and any given indexing strategy I: evaluate first each tree pattern individually, exploiting the index; then, apply the value joins on the tree pattern results thus obtained.

3.6 Concrete deployment

As outlined before, our architecture can be deployed on top of the main existing commercial cloud platforms (see Table 3.1). The concrete implementation we have used for our tests relies on AWS as of October 2012. In this Section, we describe the AWS components employed in the implementation, and discuss their role in the whole architecture.

Amazon Simple Storage Service, or S3 in short, is a file storage service for raw data. S3 stores the data in buckets identified by their name. Each bucket consists of a set of objects, each having an associated unique name (within the bucket), metadata (both system-defined and user-defined), an access control policy for AWS users and a version ID. We opted for storing the whole data set in one bucket because (i) in our setting we did not use e.g. different access control policies for different users, and (ii) Amazon states that the number of buckets used for a given dataset does not affect S3 storage and retrieval performance.

Amazon DynamoDB is a NoSQL database service for storing and querying a collection of *tables*. Each table is a collection of *items* whose size can be at most 64KB. In turn, each item contains one or more *attributes*; an attribute has a name, and one

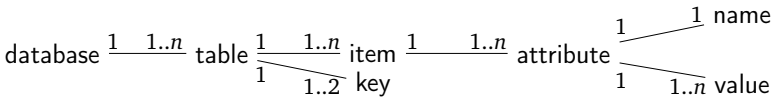


Figure 3.6: Structure of a DynamoDB database.

or several values. Each table must contain a *primary key*, which can be either (i) a single-attribute hash key with a value of at most 2KB or (ii) a composite hash-range key that combines a 2KB hash key and a 1KB range key. Figure 3.6 shows the structure of a DynamoDB database. Different items within a table may have different sets of attributes.

A table can be queried through a *get(T,k)* operation, retrieving all items in the table T having a hash key value k. If the table uses a composite hash-range key, we may use it in a call *get(T,k,c)* which retrieves all items in table T with hash key k and range key satisfying the condition c. A *batchGet* variant permits to execute 100 *get* operations through a single API request. To create an item, we use the *put(T,(a,v^+)$^+$)* operation, which inserts the attribute(s) *(a,v^+)$^+$* into a newly created item in table T; in this case, *(a,v^+)$^+$* must include the attribute(s) defining the primary key in T. If an item already exists with the same primary key, the new item completely replaces the existing one. A *batchPut* variant inserts 25 items at a time.

Multiple tables cannot be queried by a single query. The combination of query results on different tables has to be done in the application layer.

In our system, Dynamo DB is used for storing and querying indexes. Previously presented indexing strategies are mapped to DynamoDB as follows. For every strategy but *2LUPI* the index is stored in a single table, while for *2LUPI* two different tables (one for each sub-index) are used. For any strategy, an entry is mapped into one or more items. Each item has a composite primary key, formed by a hash key and a range key. The first one corresponds to the key of the index entry, while the second one is a UUID [LMS05] *global unique* identifier that can be created without a central authority, generated at indexing time. Attribute names and values of the entry are respectively stored into item attribute names and values.

Using UUIDs as range keys ensures that we can insert items in the index concurrently, from multiple virtual machines, as items with the same hash key always contain different range keys and thus cannot be overwritten. Also, using UUID instead of mapping each attribute name to a range key allows the system to reduce the number of items in the store for an index entry, and thus to improve performances at query time (we can recover all items for an index entry by means of a simple *get* operation).

Amazon Elastic Compute Cloud, or EC2 in short, provides resizable computing capacity in the cloud. Using EC2, one can launch as many virtual computer instances as desired with a variety of operating systems and execute applications on them.

AWS provides different types of instances, with different hardware specifications and prices, among which the users can choose: standard instances are well suited for most applications, high-memory instances for high throughput applications, high-

$ST^{\$}_{m,GB} = \$0.125$	$IDX^{\$}_{m,GB} = \$1.14$
$ST^{\$}_{put} = \$0.000011$	$IDX^{\$}_{put} = \$0.00000032$
$ST^{\$}_{get} = \$0.0000011$	$IDX^{\$}_{get} = \$0.000000032$
$VM^{\$}_{h,l} = \$0.34$	$QS^{\$} = \$0.000001$
$VM^{\$}_{h,xl} = \$0.68$	$egress^{\$}_{GB} = \$0.19$

Table 3.4: AWS Singapore costs as of October 2012.

CPU instances for compute-intensive applications. We have experimented with two types of standard instances, and we show in Section 3.7 the performance and cost differences between them.

Amazon Simple Queue Service, or SQS in short, provides reliable and scalable queues that enable asynchronous message-based communication between distributed components of an application over AWS. We rely on SQS heavily for circulating computing tasks between the various modules of our architecture, that is: from the front end to the virtual instances running our indexing module for loading and indexing the data; from the front end again to the instances running the query processor for answering a query, then from the query processor back to the front end to indicate that the query has been answered and thus the results can be fetched.

Concrete AWS costs vary depending on the geographic region where AWS hosts the application. Our experiments took place in the Asia Pacific (Singapore) AWS facility, and the respective prices as of September-October 2012 are collected in Table 3.4. Note that virtual machine (instance) costs are provided for two kinds of instances, "large" ($VM^{\$}_{h,l}$) and "extra-large" ($VM^{\$}_{h,xl}$).

3.7 Experimental results

This section describes the experimental environment and results that we obtained. Section 3.7.1 describes the experimental setup, Section 3.7.2 reports our performance results, and finally, Section 3.8.1 presents our cost study.

3.7.1 Experimental setup

Our experiments ran on AWS servers from the Asia Pacific region in September-October 2012. We used the (centralized) Java-based XML query processor developed within our ViP2P project [KKMZ11], implementing an extension of the algorithm of [CDZ06] to our larger subset of XQuery. On this dialect, our experiments have shown that ViP2P's performance is close to (or better than) Saxon-B v9.1 [2].

We use two types of EC2 instances for running the indexing module and query processor:

– **Large (L)**, with 7.5 GB of RAM memory and 2 virtual cores with 2 EC2 Compute Units each.

2. http://saxon.sourceforge.net/

Indexing strategy	Average extraction time (hh:mm)	Average uploading time (hh:mm)	Total time (hh:mm)
LU	0:24	1:33	2:11
LUP	0:32	3:47	4:25
LUI	0:41	2:31	3:22
2LUPI	1:13	6:30	7:46

Table 3.5: Indexing times using 8 large (L) instances.

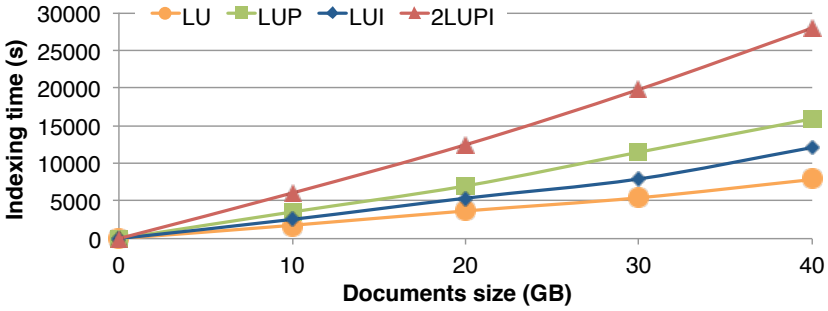


Figure 3.7: Indexing in 8 large (L) EC2 instances.

– **Extra large (XL)**, with 15 GB of RAM memory and 4 virtual cores with 2 EC2 Compute Units each.

An EC2 Compute Unit is equivalent to the CPU capacity of a 1.0-1.2 GHz 2007 Xeon processor.

To test index selectivity, we needed an XML corpus with some heterogeneity. We generated XMark [SWK+02] documents (20000 documents in all, adding up to 40 GB), using the *split option* provided by the data generator[3]. We modified a fraction of the documents to alter their *path* structure (while preserving their labels), and modified another fraction to make them "more" heterogeneous than the original documents, by rendering more elements optional children of their parents, whereas they were compulsory in XMark.

3.7.2 Performance study

XML indexing. To test the performance of index creation, the documents were initially stored in S3, from which they were gathered in batches by multiple L instances running the indexing module. We batched the documents in order to minimize the number of calls needed to load the index into DynamoDB. Moreover, we used L instances because in our configuration, DynamoDB was the bottleneck while indexing. Thus, using more powerful XL instances could not have increased the throughput.

Table 3.5 shows the time spent extracting index entries on 8 L EC2 instances and uploading the index to DynamoDB using each proposed strategy. We show the average time spent by each EC2 machine to extract the entries, the average time spent by DynamoDB to load the index data, and the total observed time (elapsed between the

3. http://www.xml-benchmark.org/faq.txt

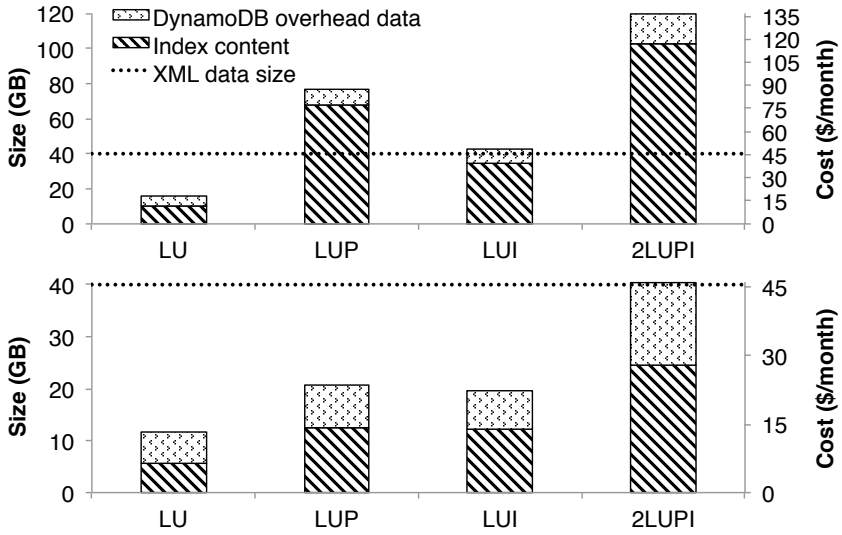


Figure 3.8: Index size and storage costs per month with full-text indexing (top) and without (bottom).

beginning and the end of the overall indexing process). As expected, the more and the larger the entries a strategy produces, the longer indexing takes. Next, Figure 3.7 shows that indexing time scales well, linearly in the size of the data for each strategy.

A different perspective on the indexing is given in Figure 3.8 which shows the size of the index entries, compared with the original XML size. In addition to the full-text indexes size, the figure includes the size for each strategy if keywords are not stored. As expected, the index for the latter strategies is quite smaller than the full-text variant. *LUP* and *2LUPI* are the larger indexes, and in particular, if we index the keywords, the index is quite larger than the data. The *LUI* index is smaller than the *LUP* one, because IDs are more compact than paths; moreover, we exploit the fact that DynamoDB allows storing arbitrary binary objects, to store compressed (encoded) sets of IDs in a single DynamoDB value. Finally, the DynamoDB space overhead (which we described in Section 3.3) is noticeable, especially if keywords are not indexed, but in both variants grows slower than the index size.

3.8 Query workload details

XML query processing. We now study the query processing performance, using 10 queries from the XMark benchmark that we depict in Figure 3.9. The queries have an average of ten nodes each; the last three queries feature value joins. Furthermore, query q_1 is very selective (point query); query q_4 uses a full-text search predicate.

Table 3.6 shows, for each query and indexing strategy, the number of documents retrieved by index look-up, the number of documents which actually contain query results, and the result size for each query. (These are obviously independent of the platform, and we provide them only to help interpret our next results.) Table 3.6

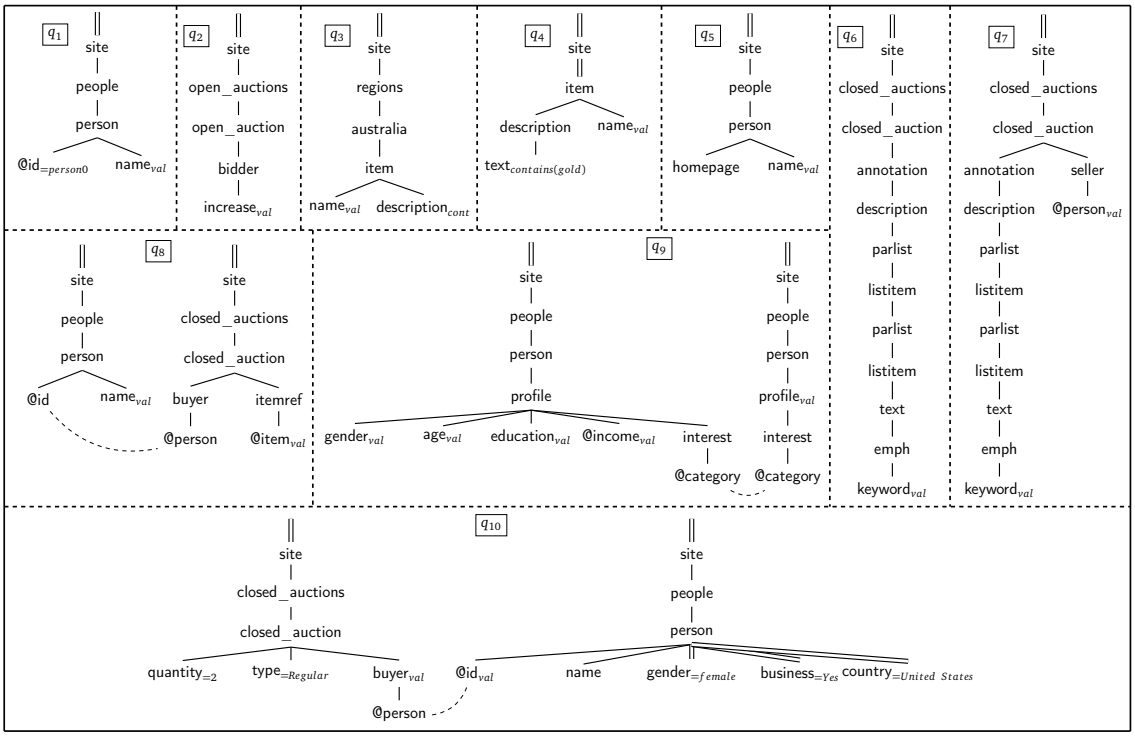
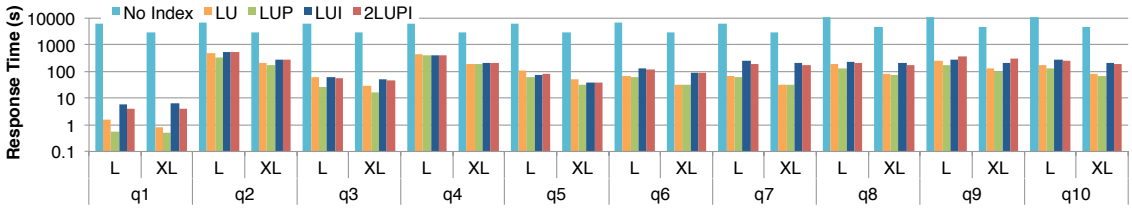


Figure 3.9: Query workload used in the experimental section.

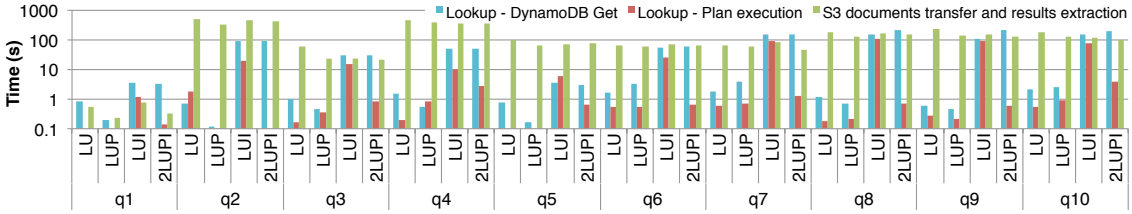
Query	# document URIs from index				# documents with results	Results size (KB)
	LU	*LUP*	*LUI*	*2LUPI*		
q_1	3	2	1	1	1	0.04
q_2	523	349	349	349	349	94000.00
q_3	144	66	33	33	33	52400.00
q_4	1089	1089	775	775	775	519.20
q_5	1115	740	370	370	370	7500.00
q_6	285	283	283	283	283	278.20
q_7	285	283	142	142	142	96.20
q_8	1400	1025	882	882	507	13800.00
q_9	1115	740	740	740	740	338800.00
q_{10}	1400	1025	512	512	116	9.10

Table 3.6: Query processing details (20000 documents).

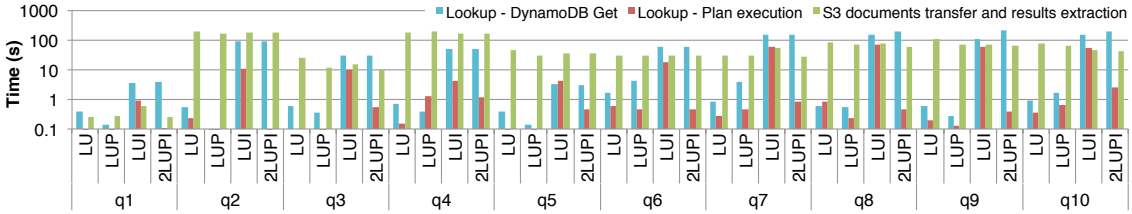
shows that *LUI* and *2LUPI* are exact for queries q_1-q_7, which are tree pattern queries (the look-up in the index returns no false positive in these cases). The imprecision of *LU* and *LUP* varies across the queries, but it may reach 200% (twice as many false positives, as there are documents with results), even for tree pattern queries like q_5. For the last three queries, featuring value joins, even *LUI* and *LUPI* may bring false positives. For these queries, Table 3.6 sums the numbers of document IDs retrieved for each tree pattern.



(a) Response time using large (L) and extra large (XL) EC2 instances on the 40 GB database



(b) Detail large (L) EC2 instance on the 40 GB database



(c) Detail extra large (XL) EC2 instance on the 40 GB database

Figure 3.10: Response time (top) and details (middle and bottom) for each query and indexing strategy.

The response times (perceived by the user) for each query, using each indexing strategy, and also without any index, is shown in Figure 3.10a; note the logarithmic *y* axis. We have evaluated the workload using L and then, separately, XL EC2 instances. We see that all indexes considerably speed up each query, by one or two orders of magnitude in most cases. Figure 3.10a also demonstrates that our strategies are able to take advantage of more powerful EC2 instances, that is, for every query, the XL running times are shorter than the times using an L instance. The strategy with the shortest evaluation time is *LUP*, which strikes a good balance between precision and efficiency; most of the time, *LU* is next, followed by *LUI* and *2LUPI* (recall again that the *y* axis is log-scale). The difference between the slowest and fastest strategy is a factor of 4 at most whereas the difference between the fastest index and no index is of 20 at least.

The charts in Figures 3.10b and 3.10c provide more insight. They split query processing time into: the time to consult the index (DynamoDB *get*), the time to run the physical plan identifying the relevant document URIs out of the data retrieved from the index, and the time to fetch the documents from S3 into EC2 and evaluate the queries there. *Importantly, since we make use of the multi-core capabilities of EC2 virtual machines, the times individually reported in Figures 3.10b and 3.10c were in fact measured in parallel*. In other words, the overall response time observed and reported in Figure 3.10a is systematically less than the sum of the detailed times reported in

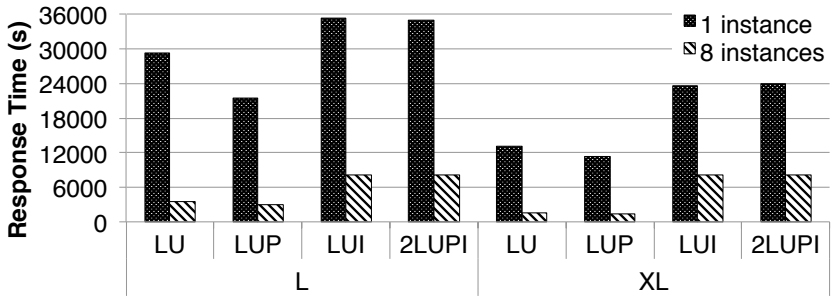


Figure 3.11: Impact of using multiple EC2 instances.

Indexing strategy	DynamoDB	EC2	S3 + SQS	Total
LU	$21.15	$5.47	$0.02	$26.64
LUP	$44.78	$11.95		$56.75
LUI	$33.47	$8.95		$42.44
2LUPI	$78.25	$21.17		$99.44

Table 3.7: Indexing costs for 40 GB using L instances.

Figures 3.10b and 3.10c.

We see that low-granularity indexing strategies (*LU* and *LUP*) have systematically shorter index look-up and index post-processing times than the fine-granularity ones (*LUI* and *2LUPI*). The times to transfer the relevant documents to EC2 and evaluate queries there, is proportional to the number of documents retrieved from the index look-ups (these numbers are provided in Table 3.6). For a given query, the document transfer + query evaluation time differs between the strategies by the factor of up to 3, corresponding to the number of documents retrieved.

Impact of parallelism. Figure 3.11 shows how the query response time varies when running multiple EC2 query processing instances. To this purpose, we sent to the front-end all our workload queries, successively, 16 times: $q_1, q_2, \ldots, q_{10}$, $q_1, q_2, \ldots$ etc. We report the running times on a single EC2 instance (no parallelism) versus running times on eight EC2 instances in parallel. We can see that more instances significantly reduce the running time, more so for L instances than for XL ones: this is because many strong instances sending indexing requests in parallel come close to saturating DynamoDB's capacity of absorbing them.

3.8.1 Amazon charged costs

We now study the costs charged by AWS for indexing the data, and for answering queries. We also consider the amortization of the index, i.e., when query cost savings brought by the index balance the cost of the index itself.

Indexing cost. Table 3.7 shows the monetary costs for indexing data according to each strategy. These costs are broken down across the specific AWS services. The most costly index to build is *2LUPI*, while the cheapest is *LU*. The combined price for S3 and SQS is constant across strategies, and is negligible compared to EC2 costs. In

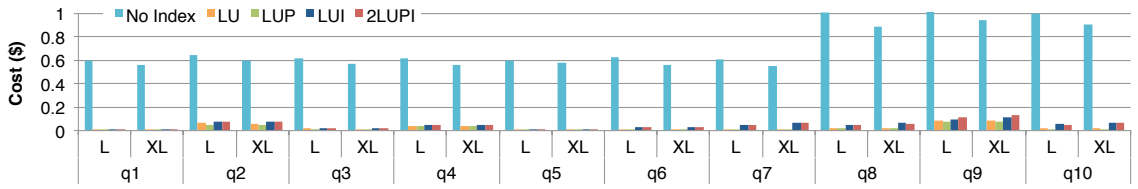


Figure 3.12: Query processing costs decomposition.

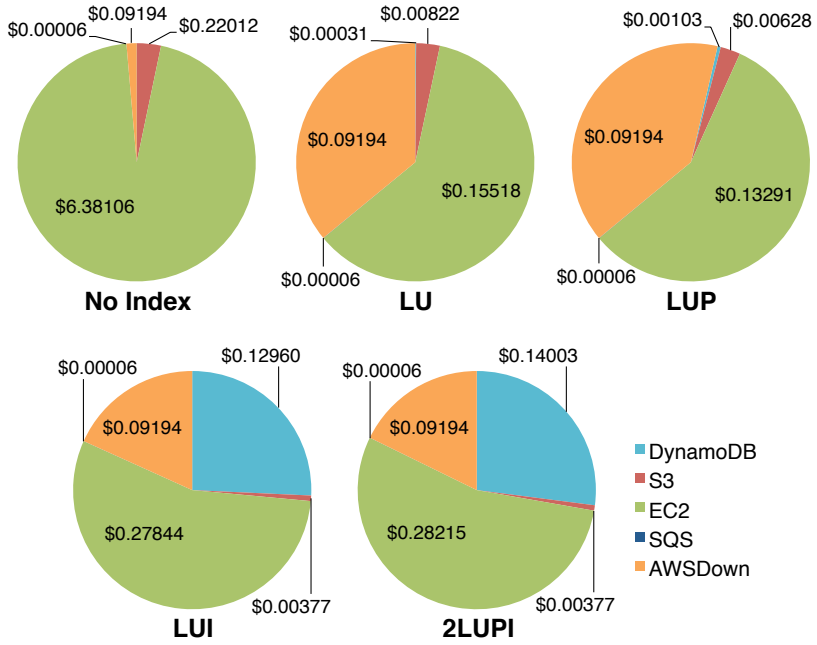


Figure 3.13: Workload evaluation cost details on an extra large (XL) instance.

turn, the EC2 cost is dominated by the DynamoDB cost in all strategies.

Finally, Figure 3.8 shows the storage cost per month of each index, which is proportional to its size in DynamoDB.

Query processing cost. Figure 3.12 shows the cost of answering each query when using no index, and when using the different indexing strategies. Note that using indexes, the cost is practically independent of the machine type. This is because (*i*) the hourly cost for an XL machine is double than that of a L machine; but at the same time (*ii*) the four cores of an XL machine allow processing queries simultaneously on twice as many documents as the two cores of L machines, so that the *cost* differences pretty much cancel each other (whereas the *time* differences are noticeable). Figure 3.12 also shows that *indexing significantly reduces monetary costs* compared to the case where no index is used; the savings vary between 92% and 97%.

To better understand the monetary costs shown in Figure 3.12, we provide the details of evaluating the query workload on an XL instance in Figure 3.13, again decomposed across the services we use, to which we add *AWSDown*, the price charged by Amazon for transferring query results out of AWS. *AWSDown* cost is the same for all strategies, since the same results are obtained. S3 cost is proportional to the selec-

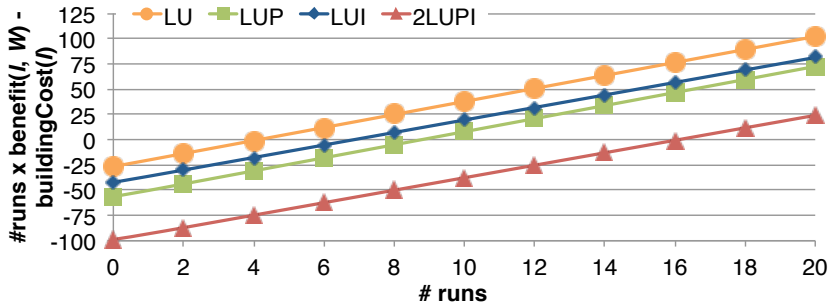


Figure 3.14: Index cost amortization for a single extra large (XL) EC2 instance.

Indexing strategy	Indexing speed (ms/MB of XML data)		Indexing cost ($/MB of XML data)	
	[CRCM12]	This chapter	[CRCM12]	This chapter
LU	7491	196	0.019	0.00067
LUP	8335	398	0.057	0.00142
LUI	12447	302	0.021	0.00106
2LUPI	11265	699	0.070	0.00249

Monthly storage cost ($/GB of XML data)

Index, [CRCM12]	Index, this chapter	Data, [CRCM12] and this chapter
0.275	1.14	0.125

Table 3.8: Indexing comparison.

tivity of the index strategy (recall Table 3.6). DynamoDB costs reflect the amount of data extracted for each strategy from the index, and finally, EC2 cost is proportional to the time necessary to answer the workload using each strategy, which means that the shorter it takes to answer a query, the lower it will be its cost. For every strategy, the cost of using EC2 clearly dominates, which is expected and desired, since this is the time actually spent processing the query.

Amortization of the index costs. We now study how indexing pays off when evaluating queries. For an indexing strategy I and workload W, we term *benefit* of I for W the difference between the monetary cost to answer W using no index, and the cost to answer W based on the index built according to I. At each run of W, we "save" this benefit, whereas we had to pay a certain cost to build I. (The index costs and benefits also depend on the data set, and increase with its size.) Figure 3.14 shows when the cumulated benefit (over several runs of the workload on a L instance) overweighs the index building cost (similarly, on a single L instance, recall Table 3.7). Figure 3.14 shows that any of our strategies allows recovering the index creation costs quite fast, i.e. just running the workload 4 times for *LU*, 8 times for *LUP* and *LUI*, and *16* times for *2LUPI* respectively. (The cost is recovered when the curves cross the $Y = 0$ axis.)

Indexing strategy	Query speed (ms/MB of XML data)		Query costs ($/MB of XML data)	
	[CRCM12]	This work	[CRCM12]	This work
LU	141	21	4.7×10^{-5}	0.6×10^{-5}
LUP	121	18	4.2×10^{-5}	0.6×10^{-5}
LUI	186	37	5.6×10^{-5}	1.3×10^{-5}
2LUPI	164	37	5.4×10^{-5}	1.3×10^{-5}

Table 3.9: Query processing comparison.

3.8.2 Comparison with previous works

The closest related work is [BFG+08], which builds database services on top of a commercial cloud, and in particular AWS. They focus on implementing transactions in the cloud with various consistency models; they present experiments on the TPC-W relational benchmark, using 10.000 products (thus, a database of 315 MB of data, *about 125 times smaller than ours*). The setting thus is quite different, but a rough comparison can still be done.

At that time, Amazon did not provide a key-value store, therefore the authors built B+ trees indexes within S3. Each *transaction* in [BFG+08] retrieves one customer record, searches for six products, and orders three of them. These are all *selective (point) queries (and updates)*, thus, they only compare with q_1 among our queries, the only one which can be assimilated to a point query (very selective path matched in few documents, recall Table 3.6). For a *transaction*, they report running times between 2.8 and 11.3 seconds, while individual queries/updates are said to last less than a second. Our q_1, running in 0.5 seconds (using one instance) on our 40 GB database of data, is thus quite competitive. Moreover, [BFG+08] reports transaction costs between $\$1.5 \times 10^{-4}$ and $\$2.9 \times 10^{-3}$, very close to our $\$1.2 \times 10^{-4}$ cost of q_1 using *LUP*.

The authors of [BFG+08] subsequently also ported their index to SimpleDB [BFG+09]. Still on TPC-W transactions on a 10.000 items database, processing with a SimpleDB index was moderately faster (by a factor of less than 2) than by using their previous S3-based one.

Further, we can also establish a comparison with our initial results presented in [CRCM12]. From a performance perspective, the main difference is that [CRCM12] stored the index within AWS' previous key-value store, namely SimpleDB. Table 3.8 compares the work presented in this chapter with [CRCM12] from the perspective of indexing, and Table 3.9 from that of querying; for a fair comparison, we report measures *per MB (or GB)* of data, as [CRCM12] reported experiments on 1 GB of XML data.

The tables show that the work presented in this chapter *speeds up indexing by one to two orders of magnitude*, all the while *indexing costs are reduced by two to three orders of magnitude*; *querying is faster (and query costs are lower) by a factor of five* (roughly) wrt [CRCM12]. The reason is that DynamoDB allows storing arbitrary binary objects as values, a feature we exploited in order to efficiently encode our

index data. Moreover, DynamoDB has a shorter response time and can handle more concurrent requests than SimpleDB.

3.8.3 Experiments conclusion

Our experiments demonstrate the feasibility and interest of our architecture based on a commercial cloud, using a distributed file system to store XML data, a key-value store to store the index, and the cloud's processing engines to index and process queries. All our indexing strategies have been shown to reduce query response time *and* monetary cost, by 2 orders of magnitude in our experiments; moreover, our architecture is capable of scaling up as more instances are added. The monetary costs of query processing are shown to be quite competitive, compared with previous similar works [BFG+08, CRCM12], and we have shown that the overhead of building and maintaining the index is modest, and quickly offset by the cost savings due to the ability to narrow the query to only a subset of the documents. In our tests, the *LUP* indexing strategy allowed for the most efficient query processing, at the expense of an index size somehow larger than the data. Further compression of the paths in the *LUP* index could probably make it even more competitive.

In our experiments, query execution based on the *LU* and *LUP* strategies is always faster than using the *LUI* and *2LUPI* strategies. We believe that cases for which *LUI* and *2LUPI* strategies behave better are those in which query tree patterns are multi-branched, highly selective and evaluated over a document set where most of the documents *only* match linear paths of the query. Such cases can be statically detected by using data summaries and some statistical information as presented for instance in [ABMP07b].

3.9 Related work

To the best of our knowledge, distributed XML indexing and query processing directly based on commercial cloud services had not been attempted elsewhere. Our workshop paper [CRCM12] presented preliminary work on different XML index-based querying strategies together with a first implementation on top of AWS, and focused on techniques to overcome Amazon SimpleDB [4] limitations for managing indexes. The follow-up paper presented in [CRCM13], which this chapter closely follows, featured several important novelties. First, it presented an implementation which relied on Amazon's new key-value store, DynamoDB, in order to ensure better performance in managing indexes, and, quite importantly, more predictable monetary cost estimation. We provide a performance comparison between both services in Section 3.7. Second, it presented a proper monetary cost model, which still remains valid in the contexts of several alternative commercial cloud services. Third, it introduced multiple optimizations in the indexing strategies. Finally, it reported about extensive

4. http://aws.amazon.com/simpledb/

experiments on a large dataset, with a particular focus on performances in terms of monetary cost.

Our work is among few to focus on cloud-based indexing for complex data. Among other previous works, it can be seen as a continuation of [BFG+08, BFG+09], which also exploited commercial clouds for fine-granularity data management, but (*i*) for relational data, (*ii*) with a stronger focus on transactions, and (*iii*) prior to the efficient key-value store we used to build indexes in this work. In turn, Stratustore [SZ10] is an extension for the Jena Semantic Web framework [Jen] that enables its interaction with Amazon SimpleDB for storing RDF data.

Alternative approaches which may permit to attain the same global goal of managing XML in the cloud comprises commercial database products, such as Oracle Database, IBM DB2 and Microsoft SQL Server. These products have included XML storage and query processing capabilities in their relational databases over the last ten years, and then have ported their servers to cloud-based architectures [BCD+11]. Differently from our framework, such systems have many functionalities beyond those for XML stores, and require non-negligible efforts for their administration, since they are characterized by complex architectures.

Another related approach is to aim at leveraging large-scale distributed infrastructures (e.g., clouds) by intra-query parallelism, as in [FLGP11] or the work that we present in Chapter 4, enabling parallelism in the processing of each query, by exploiting multiple machines. Differently, in this chapter, we consider the evaluation of one query as an atomic (inseparable) unit of processing, and focus on the horizontal scaling of the overall indexing and query processing pipeline distributed over the cloud.

Finally, some recent works related to cloud services have put special emphasis on the economic side of the cloud. For instance, the cost of multiple architectures for transaction processing on top of different commercial cloud providers is studied in [KKL10]. In this case, the authors focus on read and update workloads rather than XML processing, as in our study. On the other hand, some works have proposed models for determining the optimal *price* [KDF+11] or have studied cost amortization [KDGA11] for data-based cloud services. However, in these works, the monetary costs are studied from the perspective of the cloud providers, rather than from the user perspective, as in this chapter.

3.10 Summary

In this chapter, we have presented AMADA, an architecture for building scalable Web data warehouses by means of commercial cloud resources, which can exploit parallelism to speed up index building and query processing. We have investigated and compared through experiments several indexing strategies for XML data and shown that they achieve query processing speed-up and monetary costs reductions of several orders of magnitude within AWS.

Acknowledgements. This work has been partially funded by the KIC EIT ICT Labs

activities 11803, 11880 and RCLD 12115, as well as an Amazon research grant “AWS in Education".

Chapter 4

PAXQuery: Efficient Parallel Processing of XQuery

This chapter presents a novel approach for parallelizing the execution of complex queries over XML documents, implemented within our system PAXQuery. We provide algorithms showing how to translate such queries into plans expressed in the PArallelization ConTracts (PACT) programming model. These plans are then optimized and executed in parallel by the Stratosphere system [1]. We demonstrate the efficiency and scalability of our approach through experiments on hundreds of GB of XML data.

A short presentation of the PAXQuery system was published in a workshop [CRCM14]. The material of this chapter is being considered for publication in an international journal. Currently, PAXQuery is being integrated within Stratosphere's latest release, and will be open-sourced in the near future as an extension of the Stratosphere platform [2].

4.1 Introduction

By far the most widely adopted *implicit parallel* framework, MapReduce [DG04] features a very simple processing model consisting of two operations, *Map* which distributes processing over sets of (key, value) pairs, and *Reduce* which processes the sets of results computed by *Map* for each distinct key. While the simplicity of MapReduce is an advantage, it is also a limitation, since large data processing tasks are represented by complex programs consisting of many *Map* and *Reduce* tasks. In particular, since these tasks are conceptually basic, one often needs to write programs comprising many successive tasks, which limits parallelism. To overcome this problem, more powerful abstractions have been proposed to express massively parallel complex data processing, such as the *Resilient Distributed Datasets* [ZCD+12] or the *PArallelization ConTracts* programming model [BEH+10] (or **PACT**, in short).

1. Stratosphere has been recently accepted as an Apache incubator project and will be renamed to Apache Flink (http://incubator.apache.org/projects/flink.html).

2. http://cloak.saclay.inria.fr/research/paxquery/

In particular, PACT programs are declarative enough to take advantage of intelligent compiling techniques in order to be very efficiently evaluated. At compile time, the compiler choses an optimal strategy (plan) that maximizes parallelisation opportunities, and thus efficiency.

In a nutshell, PACT generalizes MapReduce by (i) manipulating records with any number of fields, instead of (key, value) pairs, (ii) enabling the *definition of custom parallel operators* by means of second-order functions, and (iii) allowing one parallel operator to receive as input the outputs of *several* other operators. The PACT model is part of the open-source **Stratosphere** platform [Str], which works on top of the Hadoop Distributed File System (HDFS) [Had].

In this work, we present PAXQuery, a massively parallel processor of XML queries. Given a very large collection of XML documents, evaluating a query that *navigates over these documents and also joins results from different documents* raises performance challenges, which may be addressed by parallelism. Inspired by other high-level data analytics languages that are compiled to MapReduce programs (e.g., Pig [ORS+08], Hive [TSJ+10] or Jaql [BEG+11]), PAXQuery translates XML queries into PACT plans. The main advantage of this approach is *implicit parallelism*: neither the application nor the user need to partition the XML input or the query across nodes. This contrasts with prior work [BCM+13, CLK+12, KCS11]. Further, we can rely on the Stratosphere platform for the optimization of the PACT plan and its automatic transformation into a data flow that is evaluated in parallel on top of Hadoop; these steps are explained in [BEH+10].

The contributions of this chapter are the following:

- We present a novel *methodology for massively parallel evaluation of XQuery*, based on PACT and previous research in algebraic XQuery optimization.
- We provide a *translation algorithm* from the algebraic operators required by a large powerful fragment of XQuery into operators of the PACT parallel framework. This enables parallel XQuery evaluation *without requiring data or query partitioning effort from the application*.
  Toward this goal, we first translate XML data instances (trees with node identity) into PACT nested records, to ensure XML query results are returned after the PACT manipulations of nested records.
  Second, we bridge the gap between the XQuery algebra, and in particular, many flavors of joins [DPX04, MPV09, MHM06] going beyond simple conjunctive equality joins, and PACT operators which (like MapReduce) strongly rely on grouping input records in terms of equality of their keys.
  Our translation of complex joins into PACT is of interest beyond the XQuery context, as it may enable compiling other high-level languages [BEG+11, ORS+08, TSJ+10] into PACT to take advantage of its efficiency.
- We fully implemented our translation technique into our PAXQuery platform. We present experiments demonstrating that our translation approach (i) effectively parallelizes XQuery evaluation taking advantage of the PACT framework, and (ii) scales well beyond alternative approaches for implicitly parallel XQuery evaluation, in particular as soon as joins across documents are present in the

workload.

The remainder of the chapter is organized as follows. Section 4.2 introduces the approach, while Section 4.3 provides background on XQuery and the PACT model. Section 4.4 overviews our complete solution and characterizes the XQuery algebras targeted by our translation. Section 4.5 presents the translation algorithm from algebra plans to PACT, at the core of this work. Section 4.6 describes our experimental evaluation. Section 4.7 discusses related work, before concluding this chapter.

4.2 Motivation

We illustrate the need for the parallel processing techniques developed in this chapter by means of the following example.

Example 1. Consider the following XQuery over XMark [SWK+02] documents. The query extracts the name of users, and the items of their auctions (if any):

```
let $pc := collection('people'),
    $cc := collection('closed_auctions')
for $p in $pc/site/people/person, $i in $p/@id
let $n := $p/name
let $r :=
  for $c in $cc//closed_auction, $b in $c/buyer/@person,
      $s in $c/seller/@person
  let $a := $c/itemref
  where $i = $b or $i = $s
  return $a
return <res>{$n,$r}</res>
```

◇

Suppose we need to evaluate this query over two large collections of documents (concerning people, respectively closed auctions) stored in HDFS. Evaluating the query in a massively parallel fashion as previously proposed, e.g., in [KCS11] needs the programmer to explicitly insert parallelization primitives in the query, which in turn requires time and advanced expertise.

Alternatively, one could partition the XML data and run the query as it is, over the partitioned input. Obviously, this requires the partitioning to be such that the query runs with no modification on the partitions. Partitioning strategies meeting this goal have been studied e.g., in [BCM+13, CLK+12]. Further, for complex XQuery queries like the one in Example 1, this method also necessitates input from the application designer. In particular, it requires manual decomposition of the query into (*i*) "embarrassingly parallel" subqueries which can be directly run in parallel over many documents, and (*ii*) a "recomposition" query that applies the remaining query operations.

In contrast, for the above query, PAXQuery generates *in a fully automated fashion* the PACT program shown in Figure 4.1. We outline here its functioning while on purpose omitting details, which will be introduced later on.

- The *xmlscan*('people') and *xmlscan*('closed_auctions') operators scan (in parallel) the respective collections and transform each document into a record.
- Next, the `map` operators navigate in parallel within the records thus obtained, following the query's XPath expressions, and bind the query variables.

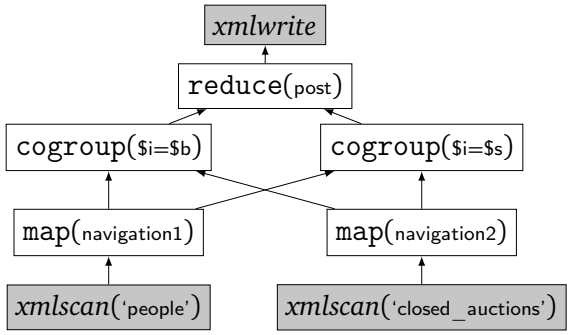


Figure 4.1: Outline of the PACT program generated by PAXQuery for the XQuery in Example 1.

- The next operators in the PACT plan (`cogroup`) go beyond MapReduce. In a nutshell, a `cogroup` can be seen as a `reduce` operator on multiple inputs: it groups together records from all inputs sharing the same key value, and then it applies a user-defined function on each group. In this example, the functions are actually quite complex (we explain them in Section 4.5). The difficulty they have to solve is to correctly express (i) the *disjunction* in the where clause of the query, and (ii) the *outerjoin* semantics frequent in XQuery: in this example, a `<res>` element must be output even for people with no auctions.
- The output of both `cogroup` operators is received by the `reduce`, which builds the final join results between people and closed_auctions.
- The last *xmlstore* builds and returns XML results.

This approach enables us to take advantage of the Stratosphere platform [Str] in order to automatically parallelize complex XML processing, expressed in a rich dialect of XQuery. In contrast, *state-of-the-art solutions require partitioning, among nodes and by hand, the query and/or the data*. Moreover, using PACT gives PAXQuery a performance advantage over MapReduce-based systems, because PACT's more expressive massively parallel operators allow more efficient query implementations.

4.3 Background

To support the presentation of our approach, we now provide background on the XQuery dialect we target (Section 4.3.1), and the PACT programming model used by Stratosphere (Section 4.3.2).

4.3.1 XQuery fragment

We consider a representative subset of the XQuery 3.0 language [W3C14b]. Our goal was to cover (i) the main navigating features of XQuery, and (ii) key constructs to express analytical style queries e.g., aggregation, explicit grouping, or rich comparison predicates. However, extensions to support other XQuery constructs e.g., *if* or *switch* expressions, can be integrated into our proposal in a straightforward manner.

Query ::= *FLWRExpr*
FLWRExpr ::= *Initial Middle* Return*
Initial ::= *For* | *Let*
Middle ::= *Initial* | *Where* | *GroupBy*
For ::= for *ForBinding* (, *ForBinding*)*
ForBinding ::= *Var* in *PathExpr*
PathExpr ::= (distinct-values)? (collection(*Uri*) | doc(*Uri*) | *Var*) *Path*
Let ::= let *LetBinding* (, *LetBinding*)*
LetBinding ::= *Var* := (*FLWRExpr* | *AggrExpr* | *PathExpr*)
AggrExpr ::= (count | avg | max | min | sum) *Var*
Where ::= where *OrExpr*
OrExpr ::= *AndExpr* (or *AndExpr*)*
AndExpr ::= *BoolExpr* (and *BoolExpr*)*
BoolExpr ::= (not)? (*Pred* | *Contains* | *Empty*)
Pred ::= *Var* (*ValCmp* | *NodeCmp*) (*Var* | *C*)
Contains ::= contains (*Var*, *C*)
Empty ::= empty (*Var*)
GroupBy ::= group by *Var* (, *Var*)*
Return ::= return (*EleConst* | (*AggrExpr* | *Var*)+)
EleConst ::= <*EName Att** (/> | (> (*EleConst* | *AggrExpr* | *Var*)* </ *EName* >))
Att ::= *AName* = "(*AggrExpr* | *Var* | *AVal*)*"
Var ::= $*VarName*

Figure 4.2: Grammar for the considered XQuery dialect.

Figure 4.2 depicts the grammar for our dialect. A query is a FLWR expression, which is a powerful abstraction that can be used for many purposes, including iterating over sequences, joining multiple documents, and performing grouping.

The initial clause of the expression is a *for* or *let*. The *for* clause iterates over the items in the sequence resulting from its associated expression, binding the variable to each item. In turn, a *let* clause binds the variable to the result of its associated expression, without iteration.

The bindings for *for* clauses are generated from an expression *PathExp*. A path is evaluated starting from the root of each document in a collection available at URI *Uri*, from the root of a single document available at URI *Uri*, or from the bindings of a previously introduced variable. *Path* corresponds to the navigational path used to locate nodes within trees; this dialect has been first identified in [MS02] under the name XPath$^{\{/,//,[]\}}$. In turn, *let* clauses may be used to bind variables to: an expression *PathExp*, a FLWR expression in turn, or an aggregation expression *AggrExpr*.

The clauses in the middle of the above grammar (*for*, *let*, *where*, or *group by*) may appear multiple times and in any order. The *where* clause supports boolean expressions (using and and or) in *disjunctive normal form* (DNF). We support two different types of elementary comparators: (*ValCmp*) compares atomic values, while (*NodeCmp*) compares nodes by their identity or by their document order. The *group by* clause groups tuples based on the value of the variables specified in the clause.

Finally, the FLWR expression ends with a *return* clause. For each tuple of bindings, the clause builds an XML forest using an element construction expression *EleConst* or a list of variables *Var+*. When we use the element construction expression, the value

Q_1
```
let $ic := collection('items')
let $i := $ic/site/regions//item
return count($i)
```

Q_2
```
let $ic := collection('items')
for $i in $ic/site/regions//item
let $l := $i/location/text()
group by $l
return <res><name>{$l}</name><num>{count($i)}</num></res>
```

Q_3
```
let $pc := collection('people'),
    $cc := collection('closed_auctions')
for $p in $pc/site/people/person, $i in $p/@id
let $n := $p/name/text()
let $a :=
  for $t in $cc/site/closed_auctions/closed_auction,
      $b in $t/buyer/@person
  where $b = $i
  return $t
return <item person="{$n}">{count($a)}</item>
```

Figure 4.3: Sample queries expressed in our XQuery grammar.

in *AVal* follows the XML naming convention for attribute values, while *AName* and *EName* follow the restrictions associated to the XML node naming conventions.

Figure 4.3 shows three sample queries from our supported dialect. Queries Q_1 and Q_2 use only one collection of documents while query Q_3 joins two collections. Further, Q_2 and Q_3 construct new XML elements while Q_1 returns the result of an aggregation over nodes from the input documents.

4.3.2 PACT framework

The PACT model [BEH+10] is a generalization of MapReduce. PACT plans are DAGs of *implicitly parallel operators*, that are optimized and translated into *explicit parallel data flows* by Stratosphere.

We introduce below the PACT data model and formalize the semantics of its operators.

Data model. PACT plans manipulate *records* of the form:

$$r = ((f_1, f_2, \ldots, f_n), (i_1, i_2, \ldots, i_k))$$

where $1 \leq k \leq n$ and:

- $(f_1, f_2, \ldots, f_n)$ is an ordered sequence of *fields* f_i. In turn, a field f_i is either an atomic value (string) or a ordered sequence $(r'_1, \ldots, r'_m)$ of records.
- $(i_1, i_2, \ldots, i_k)$ is an ordered, possibly empty, sequence of record positions in $[1 \ldots n]$ indicating the *key* fields for the record. Each of the key fields must be an atomic value.

The *key of a record* r is the concatenation of all the key fields $f_{i_1}, f_{i_2}, \ldots, f_{i_k}$. We denote by $r[i]$ and $r.key$ the field i and the key of record r, respectively. A $\perp$-record is a record whose fields consist of *null* ($\perp$) values.

We use $\mathscr{R}$ to denote the infinite domain of records.

Path indexes are needed to describe navigation through records and select record fields. A path index pi obeys the grammar $pi := j.pi \mid \epsilon$, with $j \geq 0$. *Navigation*

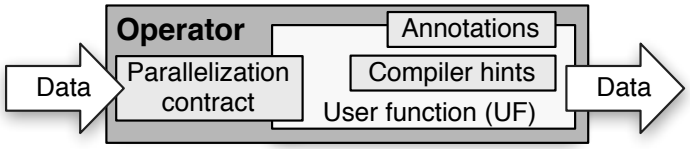


Figure 4.4: PACT operator outline.

through r along a path index j.pi first selects $r[j]$. If *pi* is empty (ϵ), then a target field is selected. Otherwise, if $r[j]$ is a list of records (the field at position j is nested), *pi* navigation is performed on each record.

Data sources and sinks are, respectively, the starting and terminal nodes of a PACT plan. The input data is stored in files; the function parameterizing data source operators specifies how to structure the data into records. In turn, data is output into files, with the destination and format similarly controlled by an output function.

Semantics. Operators are data processing nodes in a PACT plan. Each operator manipulates bags of records; we write $\{\{r_1, r_2, \ldots, r_n\}\}$ to indicate a bag of n records. From now on, for simplicity, we will call a PACT operator simply a PACT, whenever this does not cause confusion.

As Figure 4.4 shows, a PACT consists of (i) a *parallelization contract*, (ii) a *user function* (UF in short) and (iii) optional *annotations* and *compiler hints* characterizing the UF behaviour. We describe these next.

1. **Parallelization contract.** A PACT can have $k \geq 1$ inputs, each of which is a finite bag of records. The contract determines how input records are organized into *groups*. Thus, it is a function of the form:

$$c : (2^{\mathscr{R}})^k \rightarrow 2^{2^{\mathscr{R}}} \text{ for some } k \geq 1$$

   where we use the usual notation $2^{\mathscr{R}}$ to denote the power set of the set $\mathscr{R}$.

2. **User function.** The UF is executed independently over each bag of records created by the parallelization contract, therefore these executions can take place in parallel. Formally, the UF is a function of the form:

$$f : \mathscr{R}^* \rightarrow \mathscr{R}^*$$

3. Annotations and/or compiler hints may be used to enable optimizations (with no impact on the semantics), thus we do not discuss them further.

The semantics of the PACT *op* given as input k bags of records $I_1, \ldots, I_k$, with $I_i \subset \mathscr{R}$, $1 \leq i \leq k$, and having the parallelization contract c and the user function f is:

$$op(I_1, \ldots, I_k) = \bigcup_{s \in c(I_1, \ldots, I_k)} f(s)$$

In the above, c builds bags of records by grouping the input records belonging to bags $I_1, \ldots, I_k$; f is invoked on each bag produced by c, and the resulting bags are unioned.

4. Figure reproduced from [HPS+12] with permission.

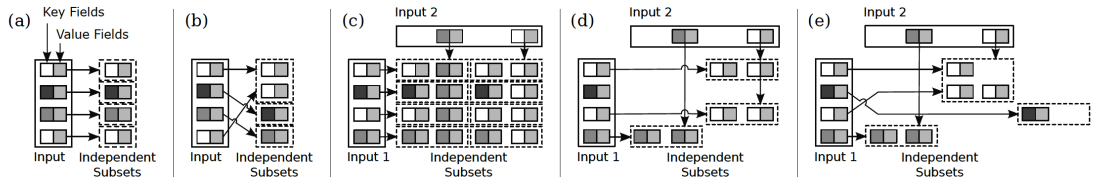


Figure 4.5: (a) `map`, (b) `reduce`, (c) `cross`, (d) `match`, and (e) `cogroup` parallelization contracts. [4]

Predefined contracts. Although the PACT model allows creating custom parallelization contracts, a set of built-in PACT operators is provided for the most common cases (see Figure 4.5). As we will show, the possibility to create expressive operators by pairing an existing input contract with a (potentially complex) user function leads to great flexibility in expressing the complex operations involved by our XQuery dialect. In the following, we detail the semantics of the contracts.

- `Map` has a single input, and builds a singleton for each input record. Formally, given the bag $I_1 \subset \mathscr{R}$ of records:

$$c_{\texttt{map}}(I_1) = \{\{\{r\} \mid r \in I_1\}\}$$

- `Reduce` also has a single input and groups together all records that share the same key. Given a bag of input records I_1:

$$c_{\texttt{reduce}}(I_1) = \{s = \{\{r_1, \ldots, r_m\}\} \mid r_1, \ldots, r_m \in I_1 \ \wedge \ r_1.key = \ldots = r_m.key \wedge \nexists r' \in I_1 \setminus s : r'.key = r_1.key)\}$$

- `Cross` builds the cartesian product of two inputs. Formally, given $I_1, I_2 \subset \mathscr{R}$:

$$c_{\texttt{cross}}(I_1, I_2) = \{\{(r_1, r_2) \mid r_1 \in I_1, r_2 \in I_2\}\}$$

- `Match` builds all pairs of records from its two inputs, which share the same key. Thus, given $I_1, I_2 \subset \mathscr{R}$:

$$c_{\texttt{match}}(I_1, I_2) = \{\{(r_1, r_2) \mid r_1 \in I_1, r_2 \in I_2 \ \wedge \ r_1.key = r_2.key\}\}$$

- `CoGroup` can be seen as a "Reduce on two inputs"; it groups the records from the both inputs, sharing the same key value. Formally, given $I_1, I_2 \subset \mathscr{R}$:

$$c_{\texttt{cogroup}}(I_1, I_2) = \{s = \{\{r_{11}, \ldots, r_{1m}, r_{21}, \ldots, r_{2j}\}\} \mid r_{11}, \ldots, r_{1m} \in I_1 \wedge r_{21}, \ldots, r_{2j} \in I_2 \wedge \forall r, r' \in s : r.key = r'.key \wedge \nexists r'' \in (I_1 \cup I_2) \setminus s : r''.key = r_{11}.key\}$$

4.4 Outline

Our approach for implicit parallel XQuery evaluation is to *translate* XQuery into PACT plans, as depicted in Figure 4.6. The central vertical stack traces the query translation steps from the top to the bottom, while at the right of each step we show the data models manipulated by that step.

First, the XQuery query is represented as an *algebraic expression*, on which multiple optimizations can be applied. XQuery translation into different algebra formalisms

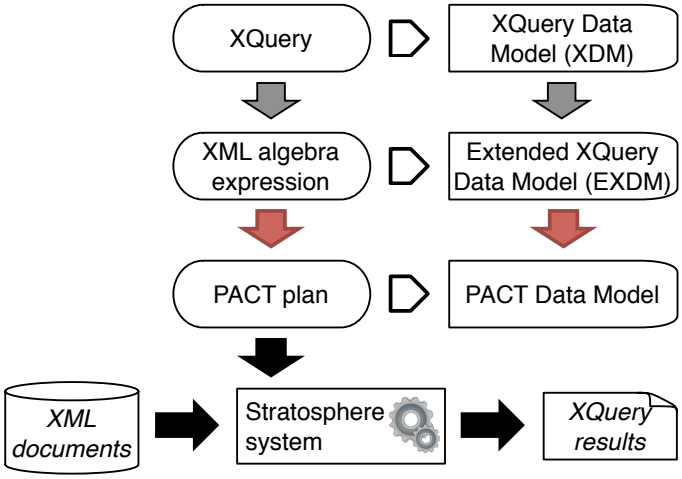


Figure 4.6: Translation process overview.

and the subsequent optimization of resulting expressions have been extensively studied [BGvK+06, RSF06, ZPR02]. In Section 4.4.1, we characterize the class of XML algebras over which our translation technique can be applied, while we present the nested-tuple data model and algebra used by our work in Section 4.4.2.

Second, *the XQuery logical expression is translated into a PACT plan*; we explain this step in detail in Section 4.5.

Finally, the Stratosphere platform receives the PACT plan, optimizes it, and turns it into a data flow that is evaluated in parallel; these steps are explained in [BEH+10] and thus they will not be detailed further in the thesis.

4.4.1 Assumptions on the XQuery algebra

Numerous logical algebras have been proposed for XQuery; see, for instance, [PWLJ04, DPX04, MPV09, RSF06]. While the language has a functional flavor, most algebras decompose the processing of a query into *operators*, such as: *navigation* (or *tree pattern matching*), which given a path (or tree pattern) query, extracts from a document *tuples* of nodes matching it; *selection*; *projection*; *join* etc.

A significant source of XQuery complexity comes from *nesting*: an XQuery expression can be nested in almost any position within another. Nested queries challenge the optimizer, as straightforward translation into nested plans leads to very poor performance.

For instance, in Figure 4.2, Q_3 contains a nested subquery `for $t ... return $t` (shown indented in the figure); let us call it Q_4 and write $Q_3 = e(Q_4)$. A naïve algebraic expression of such a query would evaluate Q_4 once per result of e in order to compute Q_3 results, which is typically inefficient.

Efficient optimization techniques translate nested XQuery into *unnested plans relying on joining and grouping* [DPX04, MPV09, MHM06]. Thus, a smarter method to represent such query is to connect the sub-plans of Q_4 and e with a *join* in the plan of Q_3; the join condition in this example is `$b=$i`. Depending on the query shape, such

decorrelating joins may be *nested* and/or *outer*.

Our goal is to complement existing engines, which translate from XQuery to an internal algebra, by an efficient compilation of this algebra into an implicit parallel framework such as PACT. This enables plugging a highly parallel back-end to an XQuery engine to improve its scalability. Accordingly, we aim to adapt to any XML query algebra satisfying the following two assumptions:

- The algebra is tuple-oriented (potentially using nested tuples).
- The algebra is rich enough to support decorrelated (unnested) plans even for nested XQuery; in particular we consider that the query plan has been unnested before we start translating it into PACT.

Three observations are of order here.

First, to express complex queries without nesting, the algebra may include *any type of joins (conjunctive/disjunctive, value or identity-based, possibly nested, possibly outer), as well as grouping*; accordingly, we must be able to translate all such operators into PACT.

Second, a tuple-based algebra for XQuery provides *border* operators for (i) creating tuples from XML trees, in leaf operators of the algebraic plan; (ii) constructing XML trees out of tuples, at the top of the algebraic plan, so that XML results can be returned.

Finally, we require no optimization but unnesting as described in [MFK01, MHM06] to be applied on the XML algebraic plan before translating it to PACT; however, any optimization *may* be applied before (and orthogonal to) our translation.

4.4.2 Algebraic representation of XQuery

The present work is based on a translation from the algebra described in [MPV09]. A methodology for translating our XQuery dialect into the algebra we consider was described in [ABM+06], and detailed through examples in [MP05]. We describe the nested tuple data model manipulated by this algebra, then present its operators.

4.4.2.1 Nested tuples data model for XML

The data model extends the W3C's XPath/XQuery data model with *nested tuples* to facilitate describing algebraic operations.
Formally, a tuple t is a list of *variable-value* pairs:

$$((\$V_1, v_1), (\$V_2, v_2), \ldots, (\$V_k, v_k))$$

where the variable names $\$V_i$ are all distinct, and each value v_i is either (i) an *item*, which can be an XML node, atomic value or $\perp$, or (ii) an *homogeneous* collection of tuples (see below).

Three flavours of *collections* are considered, namely: *lists*, *bags* and *sets*, denoted as $(t_1, t_2, \ldots, t_n)$, $\{\{t_1, t_2, \ldots, t_n\}\}$, and $\{t_1, t_2, \ldots, t_n\}$, respectively.

The *concatenation* of two tuples t_1 and t_2 is denoted by $t_1 + t_2$.

$\mathscr{A}$ ::= $construct_L$ *Operator*
Operator ::= *Scan* | *UnaryOp* | *BinaryOp*
Scan ::= *scan*
UnaryOperator ::= (*Navigation* | *Group-By* | *Flatten* | *Selection* | *Projection* | *Aggregation* | *DuplicateElimination*) *Operator*
Navigation ::= nav_e
Selection ::= sel_ρ
Projection ::= $proj_V$
Group-By ::= $grp_{G_{id},G_v,\$r}$
Flatten ::= $flat_p$
Aggregation ::= $agg_{p,a,\$r}$
DuplicateElimination ::= $dupelim_V$
BinaryOperator ::= (*CartProd* | *Join* | *LeftOuterJoin* | *NestedLeftOuterJoin*) *Operator*, *Operator*
CartProd ::= *prod*
Join ::= $join_\rho$
LeftOuterJoin ::= $ojoin^l_\rho$
NestedLeftOuterJoin ::= $nojoin^l_\rho$

Figure 4.7: XML algebraic plan grammar.

Tuple schemas. The schema S of a tuple t is a set of pairs $\{(\$V_1, S_1), \ldots, (\$V_n, S_n)\}$ where each S_i is in turn the schema of the value of the variable $\$V_i$. We use `val` to denote the type of (any) atomic value, and `node` to denote the type of XML nodes. Further, a collection of values has the schema $C\{S\}$ where C is `list`, `bag`, or `set`, depending on the kind of collection, and S is the schema of all values in the collection i.e., only *homogeneous* collections are considered.

Variable paths. Given a tuple schema $S = \{(\$V_1, S_1), \ldots, (\$V_n, S_n)\}$, a variable path p, we say p is *valid* wrt S if and only if: (i) p is ϵ, or (ii) $p = \$V_i.p'$ and $(\$V_i, S_i) \in S$ and if $p' \neq \epsilon$ then $S_i = C(S')$ and p' is valid wrt S'. If a path p is valid with respect to S, then one can follow p within any tuple t conforming to S in order to extract a value (a forest), which we denote hereafter as $t.p$.

4.4.2.2 XML algebra operators

This section provides details about algebraic operators used by the algebra considered in this work. In the following, we denote by $\mathscr{F}$ the domain of XML forests, and we denote by $\mathscr{T}$ the domain of tuples.

XML Construction ($construct_L$). The input to the operator is a collection of tuples, and from each tuple an XML forest is created: $construct_L : \mathscr{T}^* \rightarrow \mathscr{F}^*$.

The information on how to build the XML forest is specified by a list L of *construction tree patterns* (CTPs in short), attached to the *construct* operator. *For each tuple* in its input, $construct_L$ builds *one XML tree for each CTP in* L [MPV09].

Formally, Construction Tree Patterns are defined as follows.

Definition 1 (Construction Tree Pattern)**.** *A Construction Tree Pattern is a tree* $c = (V, E)$ *such that each node* $n \in V$ *is labeled with (i) a valid XML element or attribute*

Algorithm 1: XML Construction

Input : Collection of tuples T, list of CTPs L
Output: XML forest

```
1  f ← ()
2  for t ∈ T do
3      for c ∈ L do
4          r ← c.root
5          if r is not a leaf and r is labeled with a variable path p then
6              T' ← collection of tuples obtained by following p within t
7          else
8              T' ← T
9          L' ← children(r)
10         f' ← construct_L'(T')
11         if r is not optional or any variable path p_r labeling a leaf under r leads to a
           non-⊥ value within t then
12             f_r ← ()
13             if r is labeled with an element (resp. attribute) name l then
14                 f_r ← new element/attribute labeled l
15             else
16                 if r is a leaf then
17                     f_r ← t_|p
18             if f_r is not () then
19                 add f' as child of f_r
20             else
21                 f_r ← f'
22             append f_r to f
23 output f ; exit
```

name, or *(ii) a variable path p (recall Section 4.4.2.1), which is $\$V_1.p'$ where p' is in turn a variable path.*

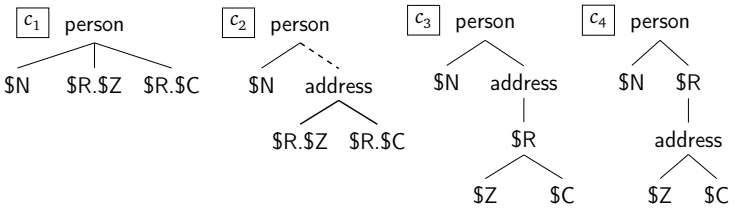
If a node n is labeled with a variable path p, and a descendant n_{desc} of n is annotated with a variable path p_{desc}, then p is a prefix of p_{desc}.

Finally, a construction subtree in c may be optional. ◇

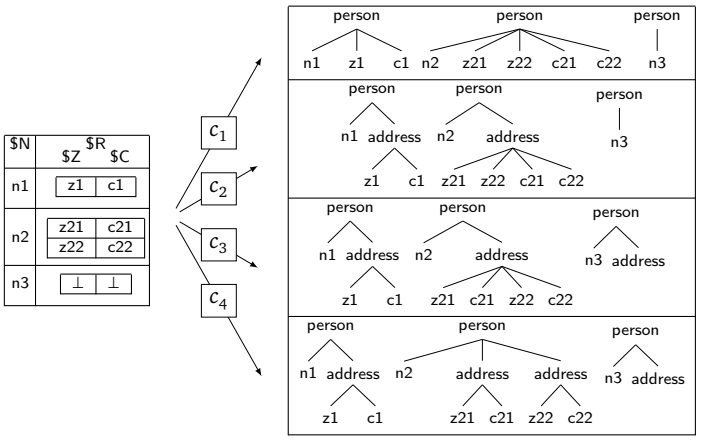
Without loss of generality, we will assume from now on that in all CTPs, the paths are valid wrt the schema of tuples in the input to the *construct* operator.

The semantics of $construct_L$ for an input collection of tuples T and a list of CTPs L is depicted in Algorithm 1. We use an XML forest f, initially empty, to gather the resulting XML.

XML content is built out of each tuple $t \in T$ during a top-down, left-to right traversal of each CTP $c \in L$; *construct* is called recursively following this order (lines 4-10). Observe that if an intermediary node in c is labeled with a variable path p, p



(a) Sample CTPs.



(b) Resulting XML forests after applying the CTPs to a collection of tuples.

Figure 4.8: Sample CTPs and corresponding XML construction results.

is followed to extract a nested collection of tuples within t (line 6), which is in turn used as input for the subsequent *construct* call (line 10). Thus, we can navigate over the nested collection of tuples to build the construction results.

Subsequently, XML content for the current node r and its children (if any) is created and appended to f iff (i) r is not the root of an optional subtree, or (ii) r is the root of an *optional* subtree but following any variable path p_r labeling a leaf under r leads to a non-$\perp$ value within t (lines 11-22).

In Figure 4.8a, we show four CTPs $c_1, \ldots, c_4$, while Figure 4.8b shows three nested tuples and the four different XML forests produced out of these three tuples by the operator $construct_{c_i, 1 \leq i \leq 4}$. We depict an *optional* construction subtree in a CTP with a dashed edge. Regardless of the construction pattern used, there are three trees in the forest, each built from one input tuple. The root of each tree is a newly created node labeled *person*, as dictated by each of the four c_is. Further, in each tree of the forest built for c_1, the children of the *person* node are deep copies [5] of the forests found in the $\$N$ attribute, respectively, in the nested $\$R.\$Z$ and $\$R.\$C$ attributes. Since in the third tuple the latter forests are empty, the third tree in the forest of c_1 only has a copy of n_3 as child. The same happens for c_2, as the subtree rooted at *address* is optional and thus it is only built if $\$R.\$Z$ or $\$R.\$C$ are not bound to $\perp$.

When XML trees are constructed based on the CTP c_4, the root node in each tree

5. Following standard XQuery semantics [W3C14b], whenever an input node needs to be output under a new parent, a deep copy of the input node is created and used in the output.

has as children (copies of) the $\$N$ nodes, as well as a newly created *address* node having the $\$Z$ and $\$C$ forests as children.

Scan (*scan*). The scan operator takes as input an XML forest and creates a tuple out of each tree in the forest: $scan : \mathscr{F} \rightarrow \mathscr{T}^*$. The semantics of the scan operator whose input is an XML forest f is the following:

$$scan(f) = \{\{\langle(\$I, d.root)\rangle \mid d \in f\}\}$$

Navigation (nav_e). XPath and XQuery may perform *navigation*, which, in a nutshell, *binds variables to the result of path traversals*. Navigation is commonly represented through *tree patterns*, whose nodes carry the labels appearing in the paths, and where some *target nodes* are also annotated with names of variables to be bound, e.g., `$pc`, `$i` etc.

The algebra we consider allows to *consolidate as many navigation operations from the same query as possible within a single navigation tree pattern*, and in particular *navigation performed outside of the* `for` *clauses* [ABM+06, DPX04, MMS07]. Large navigation patterns lead to more efficient query execution, since patterns can be matched very efficiently against XML documents; for instance, if the pattern only uses *child and descendant* edges, it can be matched in a single pass over the input, for instance using a slight extension of the algorithm described in [CDZ06].

In the spirit of *generalized tree patterns* [CJLP03], *annotated tree patterns* [PWLJ04], or *XML access modules* [ABM05], we assume a navigation (*nav*) operator parameterized by an *extended tree pattern* (ETP) supporting multiple returning nodes, child and descendant axis, and nested and optional edges. In the following we introduce ETPs formally.

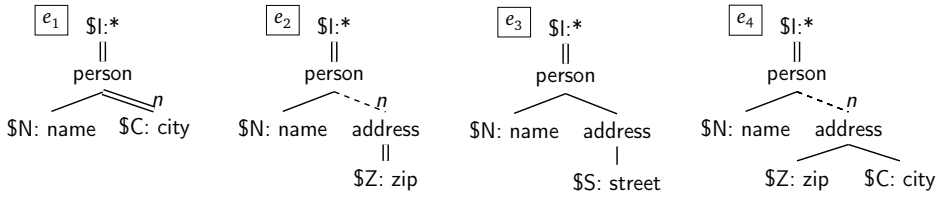
Definition 2 (Extended Tree Pattern)**.** *An Extended Tree Pattern is a tree $e = (V, E)$ where:*

- *The root $r \in V$ is labeled with a variable $\$I$.*
- *Each non-root node $n \in V$ are labeled with (i) an element/attribute name and (ii) optionally, a variable $\$V$.*
- *Each $e = (x, y) \in E$ is either a child edge from x to y, denoted by a single line, or a descendant edge from x to y, denoted by a double line. Further, optional edges are depicted with dashed lines, and nested edges are labeled with n.* ◇

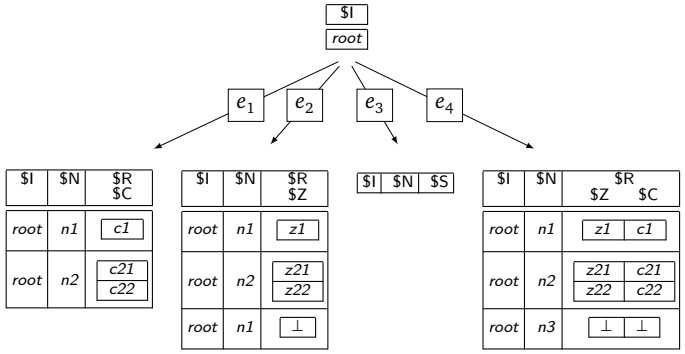
Figure 4.9a depicts some sample extended tree patterns.

Given an ETP e and an XML tree d, an *embedding* generates the tuple that results from binding the root variable of e to d and mapping the nodes of e to a collection of nodes in d. The variables of the binding tuples are ordered by the preorder traversal sequence of e. Note that if e contains *optional* edges, a mapping may be partial: nodes connected to the pattern by these edges may not be mapped, in which case the node takes the $\perp$ value.

We denote by $\varphi(e, d)$ all the embeddings from e to d. Then, we define the semantics of the navigation operator as:



(a) Sample ETPs.



(b) Resulting tuples after applying ETPs to a given tuple.

Figure 4.9: Sample ETPs and corresponding navigation results.

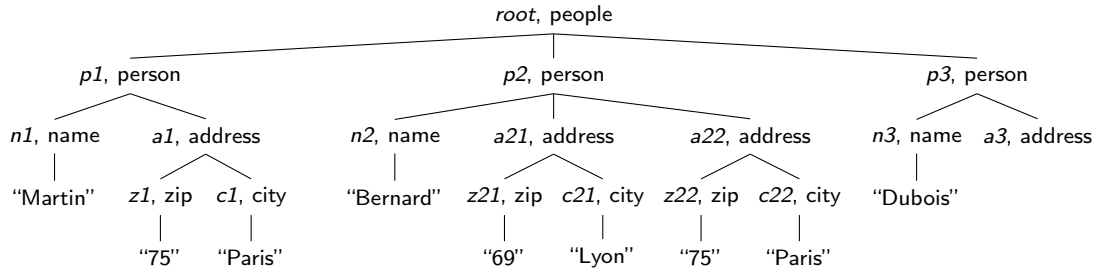


Figure 4.10: Sample XML tree.

$$nav_e(\mathscr{A}) = \bigcup_{t \in \mathscr{A}} \{\{t + t' \mid t' \in \varphi(e, t.\$I)\}\}$$

In other words, the navigation operator *nav* is parameterized by a tree pattern e, whose root is labeled with a variable $\$I$, that must appear in tuples returned by the input expression $\mathscr{A}$. The *nav* operator concatenates t successively with all tuples binding returned by $\varphi(e, t.\$I)$, for any tuple t returned by $\mathscr{A}$.

The semantics of the operator are illustrated with four examples in Figure 4.9b. Given a tuple with a variable $\$I$ bound to the XML tree shown in Figure 4.10, the navigation operator using e_1 extracts the *name* and *city* nodes of each person; observe that the variable $\$C$ is nested in $\$R$ and that the person without any *city* node does not generate any bindings. Instead, the navigation operator using e_4 generates bindings from all *person* nodes, as the subtree rooted at the *address* node is optional. The navigation result for ETPs e_2, e_3 is extracted in the similar fashion.

Group-By ($grp_{G_{id},G_v,\$r}$). The operator has three parameters: the set of group-by-id variables G_{id}, the set of group-by-value variables G_v and the result variable $\$r$.

Definition 3 (Partition function)**.** *Let $P(\mathscr{A}, G_{id}, G_v)$ be the set of tuple collections that results from partitioning the tuples output by $\mathscr{A}$, such that the tuples in a collection have id-equal values for the variables of G_{id} and equal values for the variables of G_v.*

For each collection $p \in P(\mathscr{A}, G_{id}, G_v)$, let t^p_{id} (respectively, t^p_v) be the tuple consisting of the G_{id} (respectively, G_v) variables together with their values in p. Then, the semantics of group-by operator is defined as:

$$grp_{G_{id},G_v,\$r}(\mathscr{A}) = \{\{t^p_{id} + t^p_v + \langle(\$r, p)\rangle \mid p \in P(\mathscr{A}, G_{id}, G_v)\}\}$$

Each tuple in the output of *grp* contains:
- The variables of G_{id} and G_v with their values.
- A newly introduced variable $\$r$, whose value is the group of input tuples whose G_{id} attributes are ID-equal, and whose G_v values are equal.

Flatten ($flat_p$). This operator unnests tuples in the collection referenced by p.

In the following, we specify the semantics of this operator when $p.length = 1$; the other cases can be easily worked out using the same approach. For each input tuple $t \in \mathscr{A}$, let t' (respectively, t'') be the tuple containing the variables preceding (respectively, succeeding) p in t. Further, let t_i be each of tuples contained in the collection $t_{|p}$. Then, we formalize the semantics of the operator as follows:

$$flat_p(\mathscr{A}) = \{\{t' + t^p_i + t'' \mid t' + \langle(p, t_{|p})\rangle + t'' \in \mathscr{A} \ \wedge \ t^p_i \in t_{|p}\}\}$$

Selection (sel_ρ). The selection operator is defined in the usual way based on a boolean predicate ρ to be tested on a tuple t. Formally, a selection over a stream of tuples generated by $\mathscr{A}$ is defined as:

$$sel_\rho(\mathscr{A}) = \{\{t \mid t \in \mathscr{A} \ \wedge \ \rho(t)\}\}$$

Projection ($proj_V$). The operator is defined by specifying a set of variable names $V = \{\$V_1, \ldots, \$V_k\}$ that are present at the top level of the input schema and should be retained in the output tuples. More precisely:

$$proj_V(\mathscr{A}) = \{\{\langle(\$V_1, v_1), \ldots, (\$V_k, v_k)\rangle \mid t \in \mathscr{A} \ \wedge \ \forall j \in \{1..k\}.\$V_j \in V \ \wedge \ (\$V_j, v_j) \in t\}\}$$

Aggregation ($agg_{p,a,\$r}$). The operator has three parameters: the variable path p that references the variable over whose bound values we will execute the aggregation, the aggregation operation a (recall that we support *count*, *avg*, *max*, *min* and *sum* aggregation functions), and the result variable $\$r$.

Let $A(t, p, a)$ be the result of applying the aggregation operation a on the values bound to variable path p in tuple t.

If the path p refers to a variable in a immediate nested collection, i.e. $p.length = 2$, the semantics of the aggregation operator are defined as follows.

$$agg_{p,a,\$r}(\mathscr{A}) = \{\{t + t' \mid t \in \mathscr{A} \ \wedge \ t' = \langle(\$r, A(t, p, a))\rangle\}\}$$

The semantics of the aggregation with more levels of nesting, i.e. $p.length > 2$, is straightforward.

Finally, if we want to aggregate over a non-nested variable of the input tuples, i.e. $p.length = 1$, we proceed by nesting them under a new variable to produce the correct aggregation result. Thus, the semantics is defined as follows.

$$agg_{p,a,\$r}(\mathscr{A}) = \{\{agg_{p,a,\$r}(\{t\}) \mid t = \langle(\$N, (\mathscr{A}))\rangle\}\}$$

Duplicate elimination ($dupelim_V$). The operator is defined by specifying a set of variable names V that are present at the top level of the input schema and whose bound value should be unique among the output tuples.

Recall the partition function $P(\mathscr{A}, G_{id}, G_v)$ introduced for the group-by operator. Using that function, we define the semantics of the duplicate elimination operator as:

$$dupelim_V(\mathscr{A}) = \{t_1 \mid \{\{t_1, \ldots, t_n\}\} \in P(\mathscr{A}, (), V)\}$$

Cartesian product ($prod$). The cartesian product has the standard semantics:

$$prod(\mathscr{A}_1, \mathscr{A}_2) = \{\{t_1 + t_2 \mid t_1 \in \mathscr{A}_1, t_2 \in \mathscr{A}_2\}\}$$

Join ($join_\rho$). The join relies on a boolean join predicate $\rho(t_1, t_2)$, and is defined as follows.

$$join_\rho(\mathscr{A}_1, \mathscr{A}_2) = \{\{t_1 + t_2 \mid t_1 \in \mathscr{A}_1, t_2 \in \mathscr{A}_2 \wedge \rho(t_1, t_2)\}\}$$

As stated previously, the join predicate is expressed in disjunctive normal form (DNF).

Left outer join ($ojoin^l_\rho$). Given two streams of tuples produced by $\mathscr{A}_1, \mathscr{A}_2$ and a DNF predicate ρ, $ojoin^l_\rho(\mathscr{A}_1, \mathscr{A}_2)$ returns the pairs of tuples satisfying ρ, plus the tuples from the left input without a matching right tuple. Its semantics are defined as follows:

$$\begin{aligned} ojoin^l_\rho(\mathscr{A}_1, \mathscr{A}_2) = {} & \{\{t_1 + t_2 \mid t_1 \in \mathscr{A}_1, t_2 \in \mathscr{A}_2 \wedge \rho(t_1, t_2)\}\} \cup \\ & \{\{t_1 + \bot_{\mathscr{A}_2} \mid t_1 \in \mathscr{A}_1, \nexists t_2 \in \mathscr{A}_2 : \rho(t_1, t_2)\}\} \end{aligned}$$

where $\bot_{\mathscr{A}_2}$ is a tuple having the schema of the tuples in $\mathscr{A}_2$ and $\bot$ values bound to its variables. As customary of left outer joins, the left tuples without a matching right tuple are concatenated to $\bot_{\mathscr{A}_2}$.

Nested left outer join ($nojoin^l_\rho$). The operator semantics are defined as:

$$\begin{aligned} nojoin^l_\rho(\mathscr{A}_1, \mathscr{A}_2) = {} & \{\{t_1 + \langle(\$r, (t_{21}, \ldots, t_{2n}))\rangle \mid t_1 \in \mathscr{A}_1 \wedge t_{21}, \ldots, t_{2n} \in \mathscr{A}_2 \wedge \\ & \qquad \forall k \in \{1..n\}.\rho(t_1, t_{2k})\}\} \cup \\ & \{\{t_1 + \langle(\$r, \bot_{\mathscr{A}_2})\rangle \mid t_1 \in \mathscr{A}_1, \nexists t_2 \in \mathscr{A}_2 : \rho(t_1, t_2)\}\} \end{aligned}$$

Thus, each tuple from the left input is paired with a new nested variable $\$r$, encapsulating all the matching tuples from the right-hand input. If the left tuple does not have a matching right tuple, $\$r$ must contain a tuple $\bot_{\mathscr{A}_2}$.

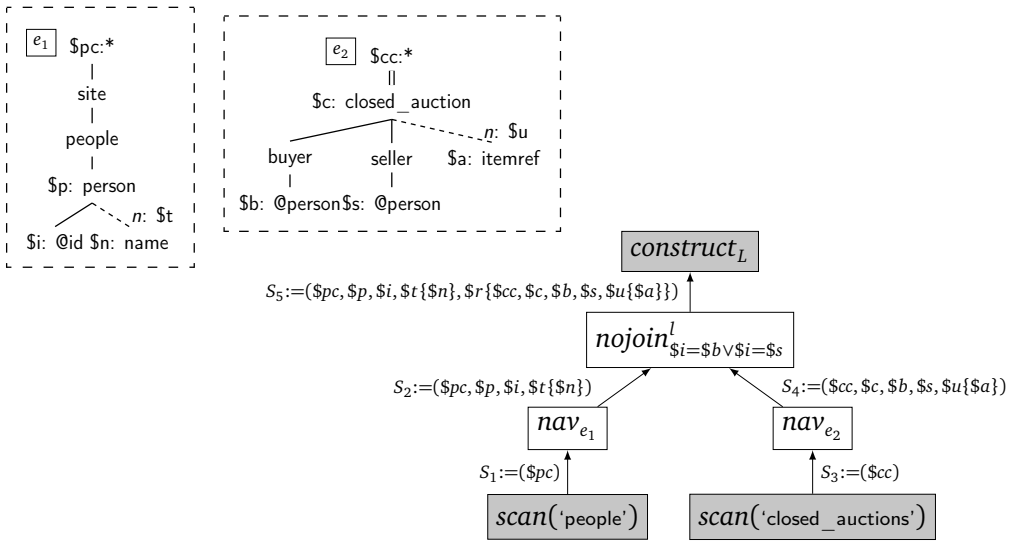


Figure 4.11: Sample logical plan for the query in Example 1.

Example 1 (*continuation*). The algebraic plan corresponding to the XQuery introduced in Section 4.2 is shown in Figure 4.11. For simplicity, we omit the variable types in the operators schema and only show the variable names. We discuss the operators starting from the leaves.

The XML *scan* operators take as input the 'people' (respectively 'closed_auctions') XML forests and create a tuple out of each tree in them. XML scan is one of the *border* operators.

Consider the ETP e_1 in Figure 4.11. The node labeled $n:name is (*i*) *optional* and (*ii*) *nested* with respect to its parent node $p:person, since by XQuery semantics: (*i*) if a given $p lacks a name, it will still contribute to the query result; (*ii*) if a given $p has several names, `let` binds them all into a single node collection. The operator nav_{e_1} concatenates each input tuple successively with all `@id` attributes (variable `$i`) and `name` elements (variable `$n`) resulting from the embeddings of e_1 in the value bound to `$pc`. Observe that variable `$n` is *nested* into variable `$t`, which did not appear in the original query; in fact, `$t` is created by the XQuery to algebra translation to hold the nested collection with values bound to `$n`. The operator nav_{e_2} is generated in a similar fashion. Therefore, in the previous query, ETPs e_1 and e_2 correspond to the following fragment:

```
for $p in $pc/site/people/person, $i in $p/@id
let $n := $p/name
let $r :=
  for $c in $cc//closed_auction, $b in $c/buyer/@person,
      $s in $c/seller/@person
  let $a := $c/itemref
```

Above the *nav* operators, we find a nested join ($nojoin^l_\rho$) on a disjunctive predicate ρ, which selects those people that appear as buyers *or* sellers in an auction.

Finally, the XML construction ($construct_L$) is the *border* operator responsible for transforming a collection of tuples to XML forests as outlined earlier in this section. *For each tuple* in its input, $construct_L$ builds *one XML tree for each CTP in L*. In our

example, L contains a single CTP that generates for each tuple an XML tree consisting of elements of the form `<res>{$n,$r}</res>`. ◇

4.5 XML algebra to PACT

Within the global approach depicted in Figure 4.6, this section describes our main contribution. First, Section 4.5.1 presents the translation of the Extended XQuery Data Model (or EXDM, in short) into the PACT Data Model. Then, Section 4.5.2 describes the translation of algebraic expressions into PACT plans, which raises the most complex technical challenges that we have to meet.

XQuery algebraic plans are translated into PACT plans recursively, operator by operator; for each XQuery operator, the translation outputs one or several PACT operators for which we need to choose (i) the *parallelization contract* (and possibly its corresponding *key fields*), and (ii) the *user function*, which together determine the PACT behavior. The hardest to translate are those algebraic operators whose input *cannot* be fragmented based on conjunctive key equalities; typical examples of this situation are disjunctive joins. This is because all massively parallel operators in PACT are based on key equality comparisons [BEH+10].

Translation rules. As in [RSF06], we use deduction rules to specify our translation. In a nutshell, a deduction rule describes *how the translation is performed when some conditions are met over the input*. Our rules rely on *translation judgments*, noted as J, J_i, and are of the form:

$$\frac{cond \;\; J_1 \;\; \ldots \;\; J_n}{J}$$

stating that the translation J (conclusion) is recursively made in terms of translations $J_1 \;\ldots\; J_n$ (premises) when the (optional) condition *cond* holds. The translation judgments J_i are optional; their absence denotes that the rule handles the "fixpoint" (start of the recursive translation).

4.5.1 Translating XML tuples into PACT records

Rules for translating instances of EXDM into those of PACT rely on translation judgments of the form:

$$t \rightarrow r$$

which reads as:

" The EXDM instance t translates into the PACT record r."

The translation rules appear in Figure 4.12, where + denotes record concatenation. *Rules produce records whose key fields are not set yet*; as we will see in Section 4.5.2, the keys are filled in by the translation.

$$\frac{v_i \to r_i \qquad i = 1 \ldots n}{((\$V_1, v_1), \ldots (\$V_n, v_n)) \to r_1 + \ldots + r_n} \quad \text{(TUPLE)}$$

$$\frac{v :: \texttt{node}}{v \to (id(v), v)} \quad \text{(XMLNODE)}$$

$$\frac{v :: \texttt{val}}{v \to (v)} \quad \text{(ATOMICVALUE)}$$

$$\frac{v :: C\{S\} \qquad v \equiv [t_1, t_2, \ldots, t_m] \qquad t_i \to r_i \qquad i = 1 \ldots m}{v \to (\,(r_1, \ldots, r_m)\,)} \quad \text{(COLLVALUE)}$$

Figure 4.12: Data model translation rules.

Rule (TUPLE) produces a record from a tuple: it translates each tuple value, and then builds the output record r by concatenating the results according to tuple order.

Three distinct rules can be triggered by rule (TUPLE). First, rule (XMLNODE) translates an XML node into a record with two fields: the first one contains the XML ID, while the second is the text serialization of the XML tree rooted at the node. In turn, rule (ATOMICVALUE) translates an XML value. Finally, rule (COLLVALUE) translates a tuple collection into a single-field record that contains the nested collection of records corresponding to the tuples in the input.

4.5.2 Translating algebraic expressions to PACT

Rules for translating an algebraic expression into a PACT plan are based on judgments of the form:

$$\mathscr{A} \Rightarrow \mathscr{P}$$

which reads as:

"The algebraic expression $\mathscr{A}$ translates into a PACT plan $\mathscr{P}$."

All rules are defined recursively over the structure of their input $\mathscr{A}$; for instance, the translation of $\mathscr{A} = sel_\rho(\mathscr{A}')$ relies on the PACT plan $\mathscr{P}'$ resulting from the translation of the smaller expression $\mathscr{A}'$, and so on.

The specific behavior of each rule is encoded in the choice of the parallelization contracts (and corresponding keys) and the user functions, so this is what we comment on below.

Overview. Table 4.1 provides an overview on the contracts used by the PACT plans resulting from our translation. First, observe that the *scan*, respectively *construct*, functionality is integrated into a data source, respectively sink, in the PACT plan. In turn, unary operators use `Map` and `Reduce` contracts depending on their semantics;

Algebra operators		**PACT operators (#)**
Scan		source (1)
Construct		sink (1)
Navigation		`map` (1)
Group-by		`reduce` (1)
Flatten		`map` (1)
Selection		`map` (1)
Projection		`map` (1)
Aggregation (on nested field)		`map` (1)
Aggregation (on top-level field)		`reduce` (1)
Duplicate elimination		`reduce` (1)
Cartesian product		`cross` (1)
Conjunctive equality join	Inner	`match` (1)
	Outer	`cogroup` (1)
	Nested outer	`cogroup` (1)
Disjunctive equality join (*n* conjunctions)	Inner	`match` (*n*)
	Outer	`cogroup` (*n*) & `reduce` (1)
	Nested outer	`cogroup` (*n*) & `reduce` (1)
Inequality join	Inner	`cross` (1)
	Outer	`cross` (1) & `reduce` (1)
	Nested outer	`cross` (1) & `reduce` (1)

Table 4.1: Algebra to PACT overview.

the implementation of their UFs is in most of the cases straightforward. Finally, the translation of the binary operators is more complex, as they have to deal efficiently with the *nested* and/or *outer* nature of some joins, which may result in multiple operators at the PACT level.

In the following, we formalize the translation algorithms and illustrate them with examples.

Preliminaries. In the translation, we denote a PACT operator by its parallelization contract c, user function f and the list K of key field positions in the PACT input. In particular:

- a unary PACT is of the form c_f^K; if $K{=}\emptyset$, for simplicity we omit it and use just c_f.
- a binary PACT is of the form $\mathsf{c}_f^{K_1,K_2}$, assuming that the key of the left input records consists of the fields K_1 and that of the right input records of K_2, respectively.

To keep track of attribute position through the translation, we use a set of *helper functions* associating to variables from S, the index positions of the corresponding fields in the PACT records. These functions are outlined in Table 4.2; we use the term *S-records* as a shortcut for records obtained by translating tuples that conform to schema S. The helper functions implementation details are quite straightforward.

Signature	Description
$S; V \mapsto_{id} F$	Given the variable paths V bound to XML nodes according to S, returns the index path positions F in S-records corresponding to the XML node IDs.
$S; V \mapsto_{v} F$	Given a list of variable paths V bound to XML nodes, atomic values or collections, according to S, returns the index path positions F of the values of those variables in S-records.
$S; V \mapsto_{id,v} F$	"Union" of the two previous functions.
$S; L \mapsto L'$	Given a list of CTPs L, returns the CTPs L' where variables are replaced with corresponding fields in S-records.
$S; e \mapsto e'$	Given an ETP e whose root is a variable in S, builds a new ETP e' rooted with the corresponding field position in S-records.
$S; \rho \mapsto \rho'$	Given an predicate ρ, builds a new predicate ρ' where variables are replaced with corresponding fields in S-records.
$S_1, S_2; \rho \mapsto \rho'$	Given a predicate ρ referencing variables in tuples in S_1 and S_2, generates a new predicate ρ' referencing field positions in S_1- and S_2-records.

Table 4.2: Auxiliary functions details.

$$\frac{\mathscr{A} \Rightarrow \mathscr{P} \qquad S_{\mathscr{A}}; L \mapsto L'}{construct_L(\mathscr{A}) \Rightarrow xmlwrite_{L'}(\mathscr{P})} \quad \text{(CONSTRUCTION)}$$

$$\frac{}{scan(f) \Rightarrow xmlscan(f)} \quad \text{(SCAN)}$$

Figure 4.13: Border operators translation rules.

4.5.2.1 Border operators translation

Figure 4.13 outlines the translation of border operators.

Rule (CONSTRUCTION) translates the logical $construct_L$ operator into a data sink that uses our output function *xmlwrite*. For each input record from $\mathscr{P}$, *xmlwrite* generates XML content using the list of construction patterns in L' and writes the results to a file. Observe that this rule relies on a $\mapsto$ auxiliary judgment to translate from the XML construction tree patterns (CTPs) L referring to an instance of our XML data model, into the patterns L' which refer to instances of the PACT data model.

Rule (SCAN) translates the logical operator $scan_f$ into a data source built up by means of our input function *xmlscan*. For each XML document in f, *xmlscan* returns a single-field record holding the content of the document.

4.5.2.2 Unary operators translation

Unary operators are translated by the rules in Figure 4.14.

Rule (NAVIGATION) uses an auxiliary judgment that translates the input ETP e into e' using $S_{\mathscr{A}}$. Navigation is applied over each record independently, and thus we use

$$\frac{\mathscr{A} \Rightarrow \mathscr{P} \qquad S_{\mathscr{A}}; e \mapsto e' \qquad f := \overline{nav}(e')}{nav_e(\mathscr{A}) \Rightarrow \mathtt{map}_f(\mathscr{P})} \quad \text{(NAVIGATION)}$$

$$\frac{\mathscr{A} \Rightarrow \mathscr{P} \qquad S_{\mathscr{A}}; G_{id} \mapsto_{id} G'_{id} \qquad S_{\mathscr{A}}; G_v \mapsto_v G'_v \qquad K := G'_{id} + G'_v \qquad f := \overline{grp}(K)}{grp_{G_{id},G_v,\$r}(\mathscr{A}) \Rightarrow \mathtt{reduce}^K_f(\mathscr{P})} \quad \text{(GROUP-BY)}$$

$$\frac{\mathscr{A} \Rightarrow \mathscr{P} \qquad S_{\mathscr{A}}; p \mapsto_v pi \qquad f := \overline{flat}(pi)}{flat_p(\mathscr{A}) \Rightarrow \mathtt{map}_f(\mathscr{P})} \quad \text{(FLATTEN)}$$

$$\frac{\mathscr{A} \Rightarrow \mathscr{P} \qquad S_{\mathscr{A}}; \rho \mapsto \rho' \qquad f := \overline{sel}(\rho')}{sel_\rho(\mathscr{A}) \Rightarrow \mathtt{map}_f(\mathscr{P})} \quad \text{(SELECTION)}$$

$$\frac{\mathscr{A} \Rightarrow \mathscr{P} \qquad S_{\mathscr{A}}; V \mapsto_{id,v} V' \qquad f := \overline{proj}(V')}{proj_V(\mathscr{A}) \Rightarrow \mathtt{map}_f(\mathscr{P})} \quad \text{(PROJECTION)}$$

$$\frac{\begin{array}{l}\mathscr{A} \Rightarrow \mathscr{P} \qquad S_{\mathscr{A}}; p \mapsto_v pi \\ \mathtt{if} \quad p.length \neq 1 \\ \mathtt{then} \quad f := \overline{agg}_n(pi, a) \quad U := \mathtt{map}_f \\ \mathtt{else} \quad K := \emptyset \quad f := \overline{agg}(pi, a) \quad U := \mathtt{reduce}^K_f\end{array}}{agg_{p,a,\$r}(\mathscr{A}) \Rightarrow U(\mathscr{P})} \quad \text{(AGGREGATION)}$$

$$\frac{\mathscr{A} \Rightarrow \mathscr{P} \qquad S_{\mathscr{A}}; V \mapsto_v K \qquad f := \overline{dupelim}}{dupelim_V(\mathscr{A}) \Rightarrow \mathtt{reduce}^K_f(\mathscr{P})} \quad \text{(DUPLELIM)}$$

Figure 4.14: Unary operators translation rules.

a PACT with a `map` contract. The UF is $\overline{nav}$, which generates new records from the (possibly partial) embeddings of e' in each input record.

Rule (GROUP-BY) translates a group-by expression into a PACT with a `reduce` contract, as the records need to be partitioned by the value of their grouping fields. The fields in K, which form the key used by the `reduce` contract, are obtained appending G'_v to G'_{id}. K is also handed to the $\overline{grp}$ UF, which creates one record from each input collection of records. The new record contains the values for each field in K, and a new field which is the collection of the input records themselves.

Example 2. The following XQuery groups together the people that share interest in the same auctions:

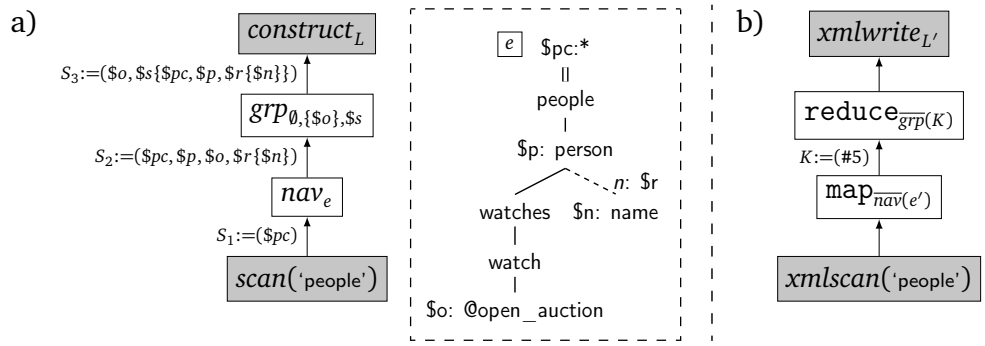


Figure 4.15: Logical expression (a) and corresponding PACT plan (b) for the query in Example 2.

```
let $pc := collection('people')
for $p in $pc//people/person, $o in $p/watches/watch/@open_auction
let $n := $p/name
group by $o
return <res><a>{$o}</a>{$n}</res>
```

The XML algebraic expression generated from this query is shown in Figure 4.15a. Using the rules in Figure 4.14, the expression is translated into the PACT plan of Figure 4.15b. Observe that the grouping variable $\$o$ is translated into field position #5, used as key for the `reduce` PACT. ◇

Rule (FLATTEN) translates a flatten expression into a `map` PACT, that applies the flattening UF $\overline{flat}$ on each input record independently. The path pi to the nested collection is obtained from p using $S_{\mathscr{A}}$.

Rule (SELECTION) produces a `map` PACT that applies the selection to each record produced by $\mathscr{P}$. Selection is performed by the $\overline{sel}$ UF, which uses the filtering condition ρ' obtained from ρ and $S_{\mathscr{A}}$.

Rule (PROJECTION) translates a projection expression into a PACT using a `map` contract. The positions V' of the fields that should be kept by the projection are obtained from V using the schema $S_{\mathscr{A}}$.

The translation of (AGGREGATION) is interesting as it can use one PACT or another, depending on the path p to the variable being aggregated. If the variable is contained in a nested collection, i.e., $p.length \neq 1$, we produce a PACT with a `map` contract; for each input record, the $\overline{agg}_n$ UF executes the aggregation operation a over the field pointed by pi and outputs a record with the aggregation results.

Otherwise, if the aggregation is executed on the complete input collection, we use a `reduce` contract wrapping the input in a single group. The $\overline{agg}$ UF creates an output record having (i) a field with a nested collection of all input records and (ii) a field with the result of executing the aggregation a over the field pointed by pi.

Finally, rule (DUPLELIM) translates a duplicate elimination expression into a PACT with a `reduce` contract. Each group handed to the UF holds the bag of records containing the same values in the fields pointed by K; the duplicate elimination UF, denoted by $\overline{dupelim}$, outputs only one record from the group.

$$\frac{\mathscr{A}_1 \Rightarrow \mathscr{P}_1 \qquad \mathscr{A}_2 \Rightarrow \mathscr{P}_2 \qquad f := \overline{concat}}{prod(\mathscr{A}_1, \mathscr{A}_2) \Rightarrow \texttt{cross}_f(\mathscr{P}_1, \mathscr{P}_2)} \quad \text{(CARTESIANPRODUCT)}$$

$$\frac{S_{\mathscr{A}_1}, S_{\mathscr{A}_2}; \rho \mapsto \rho' \qquad \mathscr{A}_1 \Rightarrow \mathscr{P}_1 \qquad \mathscr{A}_2 \Rightarrow \mathscr{P}_2 \qquad \rho' \rightharpoonup_l K_1 \qquad \rho' \rightharpoonup_r K_2 \qquad f := \overline{concat}}{join_\rho(\mathscr{A}_1, \mathscr{A}_2) \Rightarrow \texttt{match}_f^{K_1,K_2}(\mathscr{P}_1, \mathscr{P}_2)} \quad (\wedge \text{ EQUI-JOIN})$$

$$\frac{S_{\mathscr{A}_1}, S_{\mathscr{A}_2}; \rho \mapsto \rho' \qquad \mathscr{A}_1 \Rightarrow \mathscr{P}_1 \qquad \mathscr{A}_2 \Rightarrow \mathscr{P}_2 \qquad \rho' \rightharpoonup_l K_1 \qquad \rho' \rightharpoonup_r K_2 \qquad f := \overline{oconcat}_l}{ojoin^l_\rho(\mathscr{A}_1, \mathscr{A}_2) \Rightarrow \texttt{cogroup}_f^{K_1,K_2}(\mathscr{P}_1, \mathscr{P}_2)} \quad (\text{LO} \wedge \text{ EQUI-JOIN})$$

$$\frac{S_{\mathscr{A}_1}, S_{\mathscr{A}_2}; \rho \mapsto \rho' \qquad \mathscr{A}_1 \Rightarrow \mathscr{P}_1 \qquad \mathscr{A}_2 \Rightarrow \mathscr{P}_2 \qquad \rho' \rightharpoonup_l K_1 \qquad \rho' \rightharpoonup_r K_2 \qquad f := \overline{noconcat}_l}{nojoin^l_\rho(\mathscr{A}_1, \mathscr{A}_2) \Rightarrow \texttt{cogroup}_f^{K_1,K_2}(\mathscr{P}_1, \mathscr{P}_2)} \quad (\text{NLO} \wedge \text{ EQUI-JOIN})$$

Figure 4.16: Cartesian product and conjunctive equi-join translation rules.

4.5.2.3 Binary operators translation

The rules for translating these operators are the most complex. The rules themselves are depicted in Figure 4.16. We assume that the inputs $\mathscr{A}_1$ and $\mathscr{A}_2$ of the algebraic binary operator translate into the PACT plans $\mathscr{P}_1$ and $\mathscr{P}_2$.

a) Cartesian product. This operator relies on the simple *concatenation* UF, taking as input a pair of records, and outputting their concatenation: $\overline{concat}(r_1, r_2) = r_1 + r_2$.

Rule (CARTESIANPRODUCT) translates a cartesian product into a `cross` PACT with a $\overline{concat}$ UF.

b) Joins with conjunctive equality predicates. This family comprises joins on equality predicates, which can be inner equi-joins, or outer equi-joins (without loss of generality we focus on left outer joins).

b.1) Inner conjunctive equi-join. The conjunctive equi-join operator is translated by rule ($\wedge$ EQUI-JOIN), as follows. First, the predicate ρ over $\mathscr{A}_1$ and $\mathscr{A}_2$ translates into a predicate ρ' over records produced by $\mathscr{P}_1$ and $\mathscr{P}_2$. Then, the list of fields pointed by the left ($\rightharpoonup_l$), resp. right ($\rightharpoonup_r$) of the condition ρ' are extracted, and finally they are used as the keys of the generated `match` PACT.

b.2) Left outer conjunctive equi-join. In the rule (LO $\wedge$ EQUI-JOIN), the output PACT is a `cogroup` whose keys are taken from the fields of the translated join predicate ρ'. The CoGroup contract groups the records produced by $\mathscr{P}_1$ and $\mathscr{P}_2$ sharing the same key. Then, the $\overline{oconcat}_l$ UF that we describe next is applied over each group, to produce the expected result.

Definition 4 ($\overline{oconcat}_l$ user function)**.** *The left outer concatenation UF, $\overline{oconcat}_l$, of two*

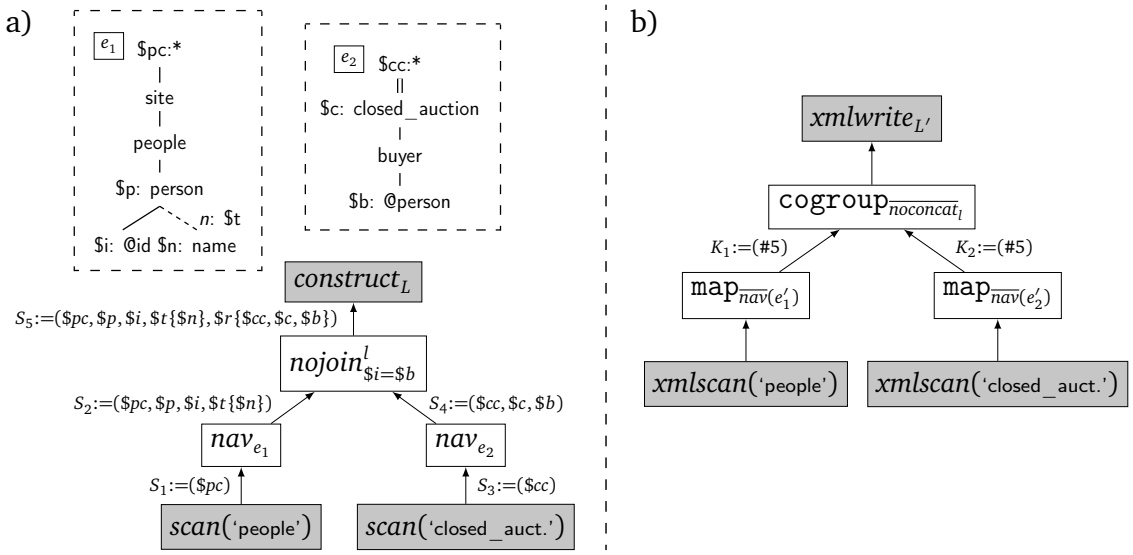


Figure 4.17: Logical expression (a) and corresponding PACT plan (b) for the query in Example 3.

record bags $\{\{r_1, \ldots, r_x\}\}$ *and* $\{\{r'_1, \ldots, r'_y\}\}$ *is defined as:*

- *If* $y \neq 0$*, the cartesian product of the two bags.*
- *Otherwise,* $\{\{r_1 + \perp', \ldots, r_x + \perp'\}\}$ *i.e., concatenate each left input record with a* $\perp$*-record having the schema (structure) of the right records.* ◇

b.3) Nested left outer conjunctive equi-join. Similar to the non-nested case, rule (NLO ∧ EQUI-JOIN) translates the nested left outer conjunctive equi-join into a `cogroup` PACT whose key is extracted from ρ'. However, we need a different UF in order to generate the desired right-hand side nested records. We define the necessary UF below.

Definition 5 ($\overline{noconcat}_l$ user function)**.** *The nested left outer concatenation UF,* $\overline{noconcat}_l$*, of the bags* $\{\{r_1, \ldots, r_x\}\}$ *and* $\{\{r'_1, \ldots, r'_y\}\}$ *is defined as:*

- *If* $y \neq 0$*,* $\{\{r_1 + (r'_1, \ldots, r'_y), \ldots, r_x + (r'_1, \ldots, r'_y)\}\}$ *i.e., nest the right set as a new field concatenated to each record from the left.*
- *Otherwise,* $\{\{r_1 + (\perp'), \ldots, r_x + (\perp')\}\}$ *i.e., add to each left record a field with a list containing a* $\perp$*-record conforming to the schema of the right records.* ◇

Example 3. The following XQuery extracts the name of users and the items that they bought (if any):

```
let $pc := collection('people'),
    $cc := collection('closed_auctions')
for $p in $pc/site/people/person, $i in $p/@id
let $n := $p/name
let $r :=
  for $c in $cc//closed_auction, $b in $c/buyer/@person
  let $a := $c/itemref
```

$$\frac{\begin{array}{c}\mathscr{A}_1 \Rightarrow \mathscr{P}_1 \qquad \mathscr{A}_2 \Rightarrow \mathscr{P}_2 \\ S_{\mathscr{A}_1}, S_{\mathscr{A}_2}; \rho \mapsto \rho' \qquad \rho' \equiv \overline{\rho'_1 \vee \rho'_2 \vee \ldots \vee \rho'_n} \\ \rho'_k \rightharpoonup_l K_{1k} \qquad \rho'_k \rightharpoonup_r K_{2k} \qquad f_k := \overline{pnjoin}(\rho', k-1) \qquad k = 1 \ldots n \\ U := \{\texttt{match}_{f_1}^{K_{11},K_{21}}, \ldots, \texttt{match}_{f_n}^{K_{1n},K_{2n}}\}\end{array}}{join_\rho(\mathscr{A}_1, \mathscr{A}_2) \Rightarrow U(\mathscr{P}_1, \mathscr{P}_2)} \quad (\vee \text{ EQUI-JOIN})$$

$$\frac{\begin{array}{c}\mathscr{A}_1 \Rightarrow \mathscr{P}_1 \qquad \mathscr{A}_2 \Rightarrow \mathscr{P}_2 \\ S_{\mathscr{A}_1}, S_{\mathscr{A}_2}; \rho \mapsto \rho' \qquad \rho' \equiv \overline{\rho'_1 \vee \rho'_2 \vee \ldots \vee \rho'_n} \\ \rho'_k \rightharpoonup_l K_{1k} \qquad \rho'_k \rightharpoonup_r K_{2k} \qquad f_k := \overline{nopnjoin_l}(\rho', k-1) \qquad k = 1 \ldots n \\ U := \{\texttt{cogroup}_{f_1}^{K_{11},K_{21}}, \ldots, \texttt{cogroup}_{f_n}^{K_{1n},K_{2n}}\} \\ S_{\mathscr{A}_1} \rightsquigarrow K \qquad f' := \overline{opost}_{l\vee}\end{array}}{ojoin_\rho^l(\mathscr{A}_1, \mathscr{A}_2) \Rightarrow \texttt{reduce}_{f'}^K(U(\mathscr{P}_1, \mathscr{P}_2))} \quad (\text{LO} \vee \text{ EQUI-JOIN})$$

$$\frac{\begin{array}{c}\mathscr{A}_1 \Rightarrow \mathscr{P}_1 \qquad \mathscr{A}_2 \Rightarrow \mathscr{P}_2 \\ S_{\mathscr{A}_1}, S_{\mathscr{A}_2}; \rho \mapsto \rho' \qquad \rho' \equiv \overline{\rho'_1 \vee \rho'_2 \vee \ldots \vee \rho'_n} \\ \rho'_k \rightharpoonup_l K_{1k} \qquad \rho'_k \rightharpoonup_r K_{2k} \qquad f_k := \overline{nopnjoin_l}(\rho', k-1) \qquad k = 1 \ldots n \\ U := \{\texttt{cogroup}_{f_1}^{K_{11},K_{21}}, \ldots, \texttt{cogroup}_{f_n}^{K_{1n},K_{2n}}\} \\ S_{\mathscr{A}_1} \rightsquigarrow K \qquad f' := \overline{nopost}_{l\vee}\end{array}}{nojoin_\rho^l(\mathscr{A}_1, \mathscr{A}_2) \Rightarrow \texttt{reduce}_{f'}^K(U(\mathscr{P}_1, \mathscr{P}_2))} \quad (\text{NLO} \vee \text{ EQUI-JOIN})$$

Figure 4.18: Disjunctive equi-join translation rules.

```
  where $i = $b
  return $a
return <res>{$n,$r}</res>
```

The query translates into the algebraic expression depicted in Figure 4.17a, while the corresponding PACT plan is shown in Figure 4.17b.

Rule (NLO $\wedge$ EQUI-JOIN) translates the nested left outer conjunctive equi-join into a PACT with a `cogroup` contract that groups together all records having the same values in the fields corresponding to $\$i$ (K_1) and $\$b$ (K_2), and applies our $\overline{noconcat}_l$ UF on them. ◇

c) Joins with disjunctive equality predicates. We focus now on disjunctive equi-joins, i.e., joins where the predicate is of the form $\rho_1 \vee \rho_2 \vee \ldots \vee \rho_n$ for some $n \geq 2$, and each ρ_i is a conjunctive predicate containing only equality comparisons.

Translating joins with disjunctive equality predicates is more involved. The reason is that PACT contracts are centered around *equality* of record fields, and thus *inherently not suited* to semantics of disjunctive conditions. To solve this mismatch, our translation relies on using more than one PACT for each operator, as we explain below.

c.1) Inner disjunctive equi-join. In rule ($\vee$ EQUI-JOIN), the predicate ρ' is generated from ρ using $S_{\mathscr{A}_1}$ and $S_{\mathscr{A}_2}$. Then, for each conjunctive predicate ρ'_k in ρ', we create a `match` whose keys are the fields participating in ρ'_k. Observe that the UFs of these

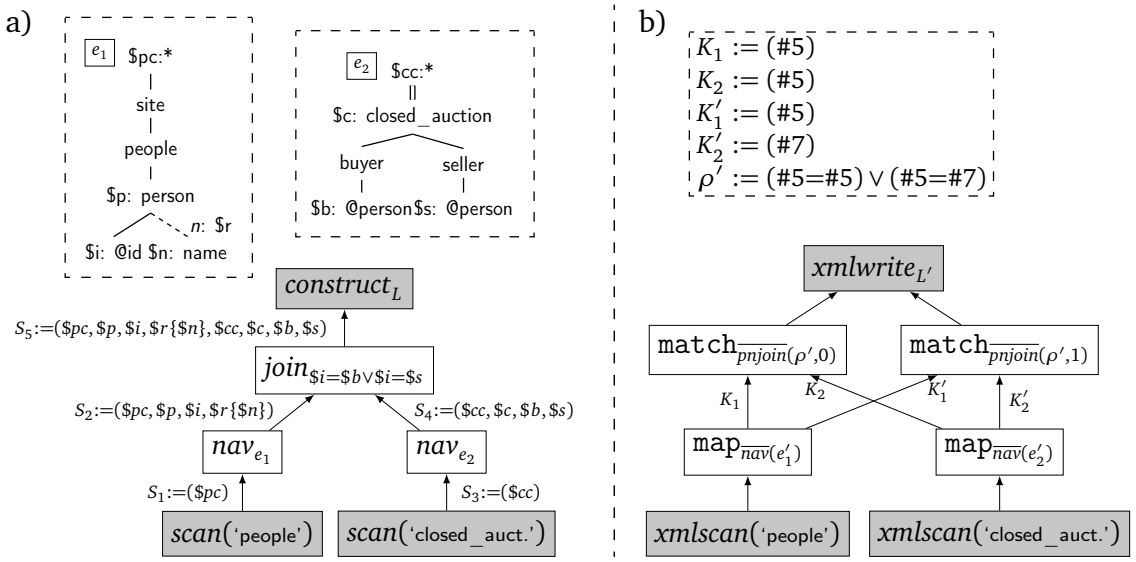


Figure 4.19: Logical expression (a) and corresponding PACT plan (b) for the query in Example 4.

`match` operators should guarantee that no erroneous duplicate is generated when the evaluation of more than one conjunctive predicates ρ'_i, ρ'_j, $i \neq j$ is true for a certain record. To that purpose, we define the new UF $\overline{pnjoin}$ below, parameterized by k and performing a partial negative join.

Definition 6 ($\overline{pnjoin}$ user function)**.** *Let $\rho' = \rho'_1 \vee \rho'_2 \vee \ldots \vee \rho'_n$ and k be an integer, with $0 \leq k < n$. Given two records r_1, r_2, the $\overline{pnjoin}(\rho', k)$ UF evaluates $\rho'_1 \vee \ldots \vee \rho'_k$ over r_1, r_2, and outputs $r_1 + r_2$ if the result is false.* ◇

Note that the UF ensures correct multiplicity of each record in the result.

Example 4. The following XQuery extracts the names of users involved in at least one auction, either as buyers or sellers:

```
let $pc := collection('people'),
    $cc := collection('closed_auctions')
for $p in $pc/site/people/person, $i in $p/@id, $c in $cc//closed_auction,
    $b in $c/buyer/@person, $s in $c/seller/@person
let $n := $p/name
where $i = $b or $i = $s
return <res>{$n}</res>
```

Rule (∨ EQUI-JOIN) translates the disjunctive equi-join into two PACTs with `match` contracts, one per disjunction. Observe that two distinct values (0 and 1) of k are used in the $\overline{pnjoin}$ UFs to prevent spurious duplicates, one for the predicate `$i=$b` and one for `$i=$s`. ◇

c.2) (Nested) left outer disjunctive equi-join. The translation of the plain and nested variants of the outer disjunctive equi-join, described by the (LO ∨ EQUI-JOIN)

and (NLO ∨ EQUI-JOIN) rules respectively, are very similar; as illustrated next, the main difference resides in the different post-processing operations they adopt. The translation of these two operators is challenging because we want to ensure parallel evaluation of each conjunctive join predicate in the disjunction, and at the same time we need to:

1. *Avoid the generation of duplicate records*. To achieve this goal, we adopt a non trivial variation of the technique used previously for disjunctive equi-join.
2. *Recognise records generated by the left hand-side expression which do not join any record coming from the right-hand side expression*. For this purpose, we use the XML node identifiers in each left hand-side record to identify it uniquely, so that, after the parallel evaluation of each conjunction, a Reduce post-processing PACT groups all resulting combinations having the same left hand-side record. If no combination exists, the left hand-side record representing a group is concatenated to a (nested) $\perp$-record conforming to the right input schema, and the resulting record is output; otherwise the output record(s) are generated from the combinations.

In the first step, we must evaluate in parallel the joins related to predicates ρ'_i. A PACT with a `cogroup` contract is built *for each conjunctive predicate* ρ'_k. Each such PACT groups together all records that share the same value in the fields pointed by ρ'_k, then applies the $\overline{nopnjoin_l}$ UF (see below) on each group, with the goal of avoiding erroneous duplicates in the result; the UF is more complex than $\overline{pnjoin}$ though, as it must handle disjunction and nesting. The $\overline{nopnjoin_l}$ function is parameterized by k, as we will use it once for each conjunction ρ'_k. Furthermore, $\overline{nopnjoin_l}$ takes as input *two* bags of records and is defined as follows, along the lines of $\overline{pnjoin}$.

Definition 7 ($\overline{nopnjoin_l}$ user function)**.** *Let* $\rho' = \rho'_1 \vee \rho'_2 \vee \ldots \vee \rho'_n$ *be a predicate where each* ρ'_i *is conjunctive. Given two input bags* $\{\{r_1, \ldots, r_x\}\}$ *and* $\{\{r'_1, \ldots, r'_y\}\}$*, the* $\overline{nopnjoin_l}(\rho', k)$ *UF is defined as follows:*

- *If the second input is empty (*$y = 0$*), return* $\{\{r_1{+}(\perp'), \ldots, r_x{+}(\perp')\}\}$ *i.e., concatenate every left input record with a field containing a nested list of one* $\perp$*-record conforming to the schema of the right input.*
- *Otherwise, for each left input record* r_i*:*
  1. *create an empty list* c_i*;*
  2. *for each* $r'_{j,1\leq j\leq y}$*, evaluate* $\rho'_1 \vee \rho'_2 \vee \ldots \vee \rho'_k$ *over* r_i *and* r'_j*, and add* r'_j *to* c_i if the result is false*;*
  3. *if* c_i *is empty, then insert into* c_i *a* $\perp$*-record with the schema of the right input;*
  4. *output* r_i *concatenated with a new field whose value is* c_i*.* ◇

The second PACT produced by the (LO ∨ EQUI-JOIN) and (NLO ∨ EQUI-JOIN) rules uses a `reduce` contract, taking as input the outputs of all the `cogroup` operators; its key consists of the XML node identifiers in each left hand-side record (we denote by

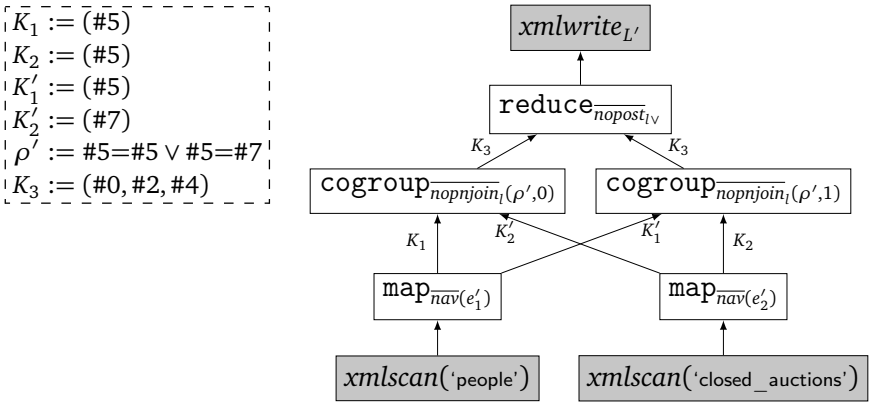


Figure 4.20: PACT plan corresponding to the logical expression in Figure 4.11.

$\rightsquigarrow$ the extraction of these fields from the schema). This amounts to grouping together the records originated from the same left input record.

Depending on the join flavor though, this last PACT uses a different UF.

- For the plain (non-nested) join (LO $\vee$ EQUI-JOIN), we use the $\overline{opost}_{l\vee}$ UF producing records with an unnested right side. The semantics is introduced in the following.
- For the nested join (NLO $\vee$ EQUI-JOIN), on the other hand, the $\overline{nopost}_{l\vee}$ UF is used to produce nested records.

The semantics of these UFs is introduced in the following.

Definition 8 ($\overline{opost}_{l\vee}$ user function)**.** *Consider an input bag of records $\{\{r_1, \ldots, r_x\}\}$. Each record $r_{i,1\leq i\leq x}$ is separated in left and right side, i.e. $r_i = r_i^l + r_i^r$. Further, r_i^r contains a single field with a nested collection of records R_i. We denote by $\overline{opost}_{l\vee}$ the post-processing UF defined as follows:*

- *If all nested collections $R_{i,1\leq i\leq x}$ contain only $\perp$-records, the UF outputs a single record $r = r_i^l + \perp'$, where r_i^l is the left side of any input record and $\perp'$ is the $\perp$-record conforming to the signature of the records in R_i.*
- *Otherwise, it flattens the nested collections $R_{i,1\leq i\leq x}$ excluding $\perp$-records, and returns the result.* $\diamond$

Definition 9 ($\overline{nopost}_{l\vee}$ user function)**.** *Consider an input bag of records $\{\{r_1, \ldots, r_x\}\}$. Each record $r_{i,1\leq i\leq x}$ is separated in left and right side, i.e. $r_i = r_i^l + r_i^r$. Further, r_i^r contains a single field with a nested collection of records R_i. We denote by $\overline{nopost}_{l\vee}$ the post-processing UF that outputs a single record $r = r_i^l + r'$, where:*

- *If all nested collections $R_{i,1\leq i\leq x}$ contain only $\perp$-records, r' contains a field with a nested collection with a $\perp$-record conforming to the signature of the records in R_i.*
- *Otherwise, r' contains a field with a nested collection of the records contained in $R_{i,1\leq i\leq x}$, excluding $\perp$-records.* $\diamond$

$$\frac{\begin{array}{cc}\mathscr{A}_1 \Rightarrow \mathscr{P}_1 & \mathscr{A}_2 \Rightarrow \mathscr{P}_2 \\ S_{\mathscr{A}_1}, S_{\mathscr{A}_2}; \rho \mapsto \rho' & f := \overline{pnjoin}(\rho')\end{array}}{join_\rho(\mathscr{A}_1, \mathscr{A}_2) \Rightarrow \texttt{cross}_f(\mathscr{P}_1, \mathscr{P}_2)} \quad \text{(INEQUI-JOIN)}$$

$$\frac{\begin{array}{cc}\mathscr{A}_1 \Rightarrow \mathscr{P}_1 & \mathscr{A}_2 \Rightarrow \mathscr{P}_2 \\ S_{\mathscr{A}_1}, S_{\mathscr{A}_2}; \rho \mapsto \rho' & f := \overline{ojoin_l}(\rho') \\ S_{\mathscr{A}_1} \rightsquigarrow K & f' := \overline{opost}_l\end{array}}{ojoin^l_\rho(\mathscr{A}_1, \mathscr{A}_2) \Rightarrow \texttt{reduce}^K_{f'}(\texttt{cross}_f(\mathscr{P}_1, \mathscr{P}_2))} \quad \text{(LO INEQUI-JOIN)}$$

$$\frac{\begin{array}{cc}\mathscr{A}_1 \Rightarrow \mathscr{P}_1 & \mathscr{A}_2 \Rightarrow \mathscr{P}_2 \\ S_{\mathscr{A}_1}, S_{\mathscr{A}_2}; \rho \mapsto \rho' & f := \overline{ojoin_l}(\rho') \\ S_{\mathscr{A}_1} \rightsquigarrow K & f' := \overline{nopost}_l\end{array}}{nojoin^l_\rho(\mathscr{A}_1, \mathscr{A}_2) \Rightarrow \texttt{reduce}^K_{f'}(\texttt{cross}_f(\mathscr{P}_1, \mathscr{P}_2))} \quad \text{(NLO INEQUI-JOIN)}$$

Figure 4.21: Inequi-join translation rules.

Example 1 (*continuation*). Our algorithms translate the algebraic expression shown in Figure 4.11 into the PACT plan depicted in Figure 4.20. This plan is the same as the one that has been shown in less detail in Figure 4.1.

Rule (NLO ∨ EQUI-JOIN) translates the nested left outer disjunctive equi-join into (i) two PACTs with `cogroup` contracts, one for each disjunction, and (ii) a PACT with a `reduce` contract that groups together records originating from the same left-hand side record, i.e., K_3 holds field positions #0, #2, #4, which contain the XML node identifiers of $\$pc, \$p, \$i$, respectively. ◇

d) Inequality joins. Our XQuery fragment also supports joins with inequality conditions. In this case, the translation uses `cross` contracts. Further, just like for joins with disjunctive predicates, the non-nested and nested outer variants of the inequi-join require more than one PACT. We depict the corresponding translation rules in Figure 4.21. In the following, we explain the translation of this flavor of joins.

d.1) Inner inequi-join. Rule (INEQUI-JOIN) generates a PACT with a `cross` contract. The predicate ρ is transformed into ρ', which is equivalent but replaces the EXDM variables by positions in the PACTs records. Then the $\overline{pnjoin}$ UF introduced in the following is applied over each pair of records.

Definition 10 ($\overline{pnjoin}$ user function)**.** *Given two records r_1, r_2 and a predicate ρ', the $\overline{pnjoin}(\rho')$ UF evaluates ρ' over r_1, r_2, and outputs $r_1{+}r_2$ if it evaluates to true.* ◇

d.2) (Nested) left outer inequi-join. Similarly with the case of disjunctive equality predicates, the translations of the non-nested and nested variant of the outer inequi-join, described by the (LO INEQUI-JOIN) and (NLO INEQUI-JOIN) rules respectively, resemble each other.

The translation of the non-nested and nested left outer inequi-join results in two steps. The first step consists of a PACT with a `cross` contract. The UF of the PACT is $\overline{ojoin}_l$, a traditional left outer join, that we introduce in the following.

Definition 11 ($\overline{ojoin}_l$ user function)**.** *Given two records r_1, r_2 and a predicate ρ', the $\overline{ojoin}_l(\rho')$ UF evaluates ρ' over r_1, r_2, and:*

- *If it evaluates to true, outputs $r_1{+}r_2$.*
- *Otherwise, it outputs $r_1{+}\perp_2$, where $\perp_2$ is the $\perp$-record that conforms to the signature of r_2.* ◇

The last PACT resulting from both translation rules uses a `reduce` contract that groups together the records originated from the same left hand-side record. In the plain variant, the UF is $\overline{opost}_l$ that produces unnested records; otherwise, the PACT uses the $\overline{nopost}_l$ UF. We introduce both UFs in the following.

Definition 12 ($\overline{opost}_l$ user function)**.** *Consider an input bag of records $\{\{r_1, \ldots, r_x\}\}$. Each record $r_{i,1\leq i\leq x}$ is separated in left and right side, i.e. $r_i = r_i^l{+}r_i^r$. We denote by $\overline{opost}_l$ the post-processing UF which:*

- *If $r^r_{i,1\leq i\leq x}$ are all $\perp$-records, it outputs one of them.*
- *Otherwise, it returns every $r_{i,1\leq i\leq x}$ where r_i^r is not a $\perp$-record.* ◇

Definition 13 ($\overline{nopost}_l$ user function)**.** *Consider an input bag of records $\{\{r_1, \ldots, r_x\}\}$. Each record $r_{i,1\leq i\leq x}$ is separated in left and right side, i.e. $r_i = r_i^l{+}r_i^r$. We denote by $\overline{nopost}_l$ the post-processing UF that outputs a single record $r = r_i^l{+}r'$, where:*

- *If $r^r_{i,1\leq i\leq x}$ are all $\perp$-records, r' contains a field with a nested collection with a $\perp$-record that conforms to the signature of r_i^r.*
- *Otherwise, r' contains a field with a nested collection with every $r^r_{i,1\leq i\leq x}$ that is not a $\perp$-record.* ◇

Example 5. Consider the following XQuery that extracts the name of users and (if any) the items they bought that were valued more than their monthly incoming:

```
let $pc := collection('people'),
 $cc := collection('closed_auctions')
for $p in $pc/site/people/person
let $n := $p/name
let $r :=
  for $c in $cc//closed_auction, $i in $p/@id, $b in $c/buyer/@person,
      $x in $p/profile/@income, $y in $c/price
  let $a := $c/itemref
  where $i = $b and $x < $y
  return $a
return <res>{$n,$r}</res>
```

The XML algebra expression generated from this query is shown in Figure 4.22a. Using the rule (LO INEQUI-JOIN) in Figure 4.21, the algebraic expression corresponding to the query is translated into the PACT plan depicted in Figure 4.22b. ◇

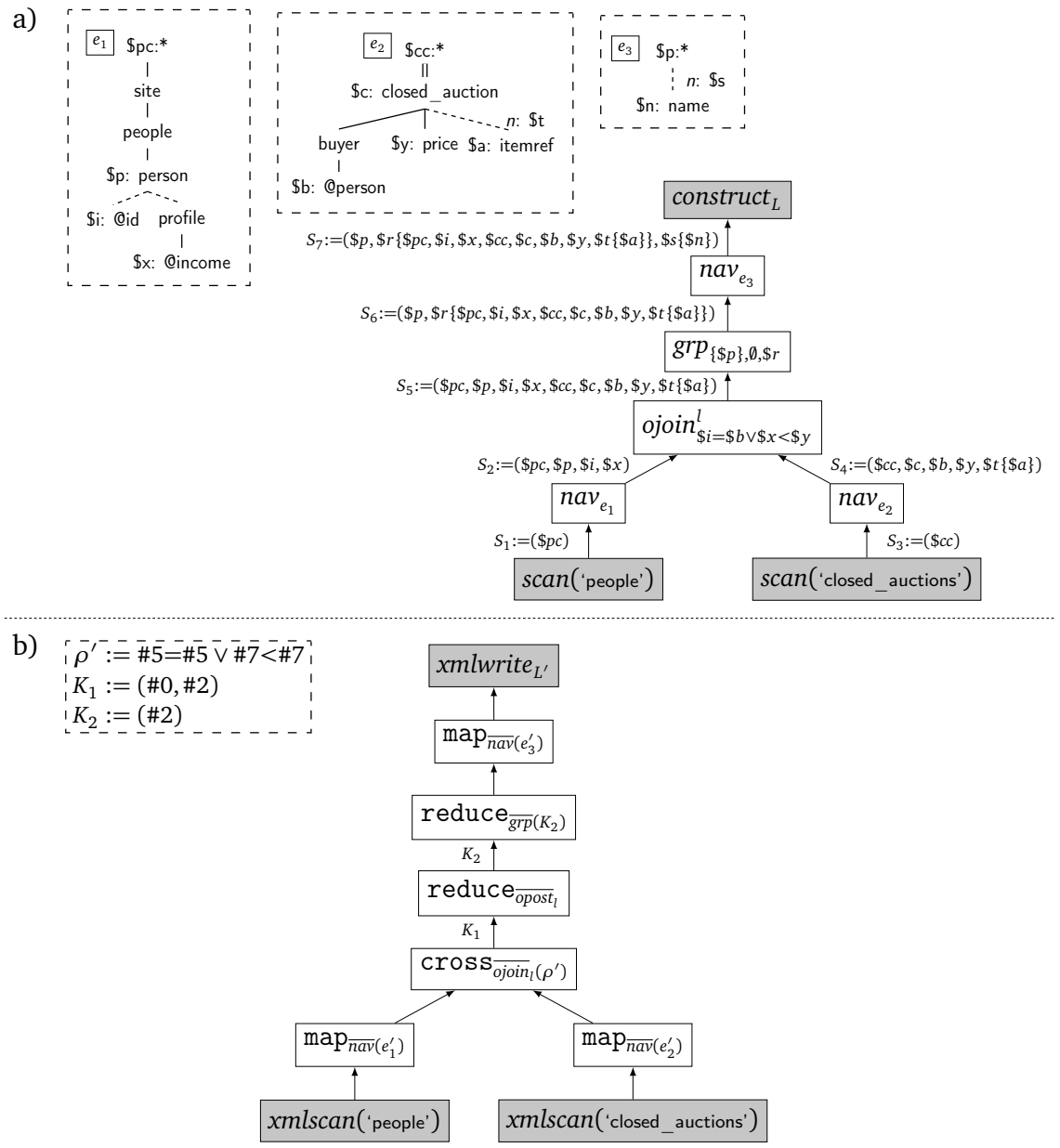


Figure 4.22: Logical expression (a) and corresponding PACT plan (b) for the query in Example 5.

Syntactically complex translation vs. performance. Clearly, complex joins such as those considered in point **c)** above could be translated into a single `cross` PACT over the pairs of records as in **d)**. However, this would be less efficient and scale poorly, since it entails a number of comparisons quadratic in the input size. The interest of the more complex, alternative translation procedure we propose is confirmed by our experiments.

4.6 Experimental evaluation

We implemented our PAXQuery translation approach in Java 1.6, and relied on the Stratosphere platform [Str] supporting PACT. The source code amounts to about 27000 lines and 170 classes. We first describe the experimental setup, and then present our results.

Experimental setup. The experiments run in a cluster of 8 nodes on an 1GB Ethernet. Each node has a 2.93GHz Quad Core Xeon processor, 16GB RAM and two 600GB SATA hard disks and runs Linux CentOS 6.4. PAXQuery is built on top of Stratosphere 0.2.1; it stores the XML data in HDFS 1.1.2.

XML data. We used XMark generated synthetic XML documents [SWK+02]. To study queries joining several documents, we used the `split` option of the XMark generator to create four collections of XML documents, each containing a specific type of XMark subtrees: *users* (10% of the dataset size), *items* (50%), *open auctions* (25%) and *closed auctions* (15%). We used datasets of up to **272GB** as detailed below.

All documents are simply stored in HDFS (which replicates them three times), that is, *we do not control the distribution/allocation of documents over the nodes*.

XML queries. We used a subset of XMark queries from our XQuery fragment, and added queries with features supported by our dialect but absent from the original XMark, e.g., joins on disjunctive predicates; all queries are detailed in Appendix A.

Table 4.3 outlines the queries: the collection(s) that each query carries over, the corresponding XML algebraic operators and their numbers of occurrences, and the parallelization contracts used in the plan generated by our translation framework. *Queries q_9-q_{14} all involve value joins, which carry over thousands of documents arbitrarily distributed across the HDFS nodes*.

4.6.1 PAXQuery scalability

Our first goal is to check that PAXQuery brings to XQuery evaluation the desired benefits of implicit parallelism. For this, we fixed a set of queries, generated 11.000 documents (**34GB**) per node, and varied the number of nodes from **1** to **2, 4, 8** respectively; the total dataset size increases accordingly in a linear fashion, up to **272GB**.

Figure 4.23 shows the response times for each query. Queries q_1-q_6 navigate in the input document according to a given navigation pattern of 5 to 14 nodes; each translates into a `map` PACT, thus their response time follows the the size of the input. These queries scale up well; we see a moderate overhead in Figure 4.23 as the data volume and number of nodes increases.

Queries q_7 and q_8 apply an aggregation over all the records generated by a navigation. For both queries, the navigation generates nested records and the aggregation consists on two steps. The first step goes over the nested fields in each input record, and thus it uses a `map` contract. The second step is executed over the results of the first. Therefore, a `reduce` contract that groups together all records coming from the

Query	Collections	Algebra operators (#)	Parallelization contracts (#)
q_1	*users*	Navigation (1)	`map` (1)
q_2	*items*	Navigation (1)	`map` (1)
q_3	*items*	Navigation (1)	`map` (1)
q_4	*closed auctions*	Navigation (1)	`map` (1)
q_5	*closed auctions*	Navigation (1)	`map` (1)
q_6	*users*	Navigation (1)	`map` (1)
q_7	*closed auctions*	Navigation (1) Aggregation (2)	`map` (2) `reduce` (1)
q_8	*items*	Navigation (1) Aggregation (2)	`map` (2) `reduce` (1)
q_9	*users* *closed auctions*	Navigation (2) Projection (1) Group-by/aggregation (1) Conj. equi-join (1)	`map` (3) `reduce` (1) `match` (1)
q_{10}	*users* *items* *closed auctions*	Navigation (3) Projection (2) NLO conj. equi-join (2)	`map` (5) `cogroup` (2)
q_{11}	*users*	Navigation (2) Projection (1) Duplicate elimination (1) NLO conj. equi-join (1)	`map` (3) `reduce` (1) `cogroup` (1)
q_{12}	*users* *closed auctions*	Navigation (2) Projection (1) NLO conj. equi-join/aggregation (1)	`map` (3) `cogroup` (1)
q_{13}	*users* *closed auctions*	Navigation (2) Projection (1) NLO disj. equi-join (1)	`map` (3) `reduce` (2) `cogroup` (2)
q_{14}	*users* *open auctions*	Navigation (2) Projection (1) NLO θ-join (1)	`map` (3) `reduce` (2) `cross` (1)

Table 4.3: Query details.

previous operator is used. Since the running time is dominated by the `map` PACTs which parallelize very well, q_7 and q_8 also scale up well.

Queries q_9-q_{12} involve conjunctive equi-joins over the collections. Query q_{13} executes a NLO disjunctive equi-join, while q_{14} applies a NLO θ-join. We notice a very good scaleup for q_9-q_{13}, whose joins are translated in many PACTs (recall the rules in Figure 4.18). In contrast, q_{14}, which translates into a `cross` PACT, scales noticeably less well. *This validates the interest of translating disjunctive equi-joins into many PACTs* (as our rules do), *rather than into a single* `cross`, since, despite parallelization, it fundamentally does not scale.

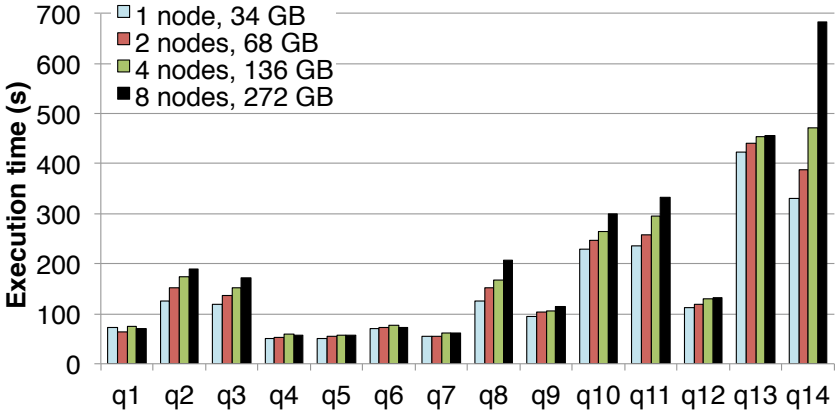


Figure 4.23: PAXQuery scalability evaluation.

Query	Evaluation time (seconds)			
	BaseX	Saxon-PE	Qizx/open	PAXQuery
q_1	206	145	90	**72**
q_2	629	~~OOM~~	~~OOM~~	**125**
q_3	600	~~OOM~~	~~OOM~~	**120**
q_4	189	~~OOM~~	84	**51**
q_5	183	125	183	**51**
q_6	233	162	109	**70**
q_7	181	111	88	**54**
q_8	599	~~OOM~~	~~OOM~~	**126**
q_9	~~TO~~	~~OOM~~	~~OOM~~	**94**
q_{10}	~~OOM~~	~~OOM~~	~~OOM~~	**229**
q_{11}	~~TO~~	~~TO~~	~~TO~~	**236**
q_{12}	~~TO~~	~~OOM~~	~~OOM~~	**113**
q_{13}	~~TO~~	~~OOM~~	~~OOM~~	**424**
q_{14}	~~OOM~~	~~OOM~~	~~OOM~~	**331**

Table 4.4: Query evaluation time (1 node, 34GB).

4.6.2 Comparison against other processors

To evaluate the performance of our processor against existing alternatives, we started by comparing it *on a single node* with other popular centralized XQuery processors. The purpose is to validate our choice of an XML algebra as outlined in Section 4.4.2 as input to our translation, by demonstrating that *single-site* query evaluation based on such an algebra is efficient. For this, we compare our processor with BaseX 7.7 [Bas], Saxon-PE 9.4 [Sax] and Qizx/open 4.1 [Qiz], on a dataset of 11000 XML documents (**34GB**).

Table 4.4 shows the response times for each query and processor; the shortest time is shown in bold, while ~~OOM~~ stands for *out of memory*, and ~~TO~~ for *timeout* (above 2 hours). In Table 4.4, we identify two query groups.

– In the first group, q_1-q_8 do not feature joins. While the performance varies across systems, only BaseX and PAXQuery are able to run all these queries.

PAXQuery outperforms other systems because, compiled in PACT, it is able to exploit the multicore architecture.

– In the second group, queries q_9-q_{14} join across the documents. None of the competing XQuery processors completes their evaluation, while PAXQuery executes them quite fast. For these, the usage of outer joins and multicore parallelization are key to this good performance behavior.

We next compare our system with other *alternatives for implicitly parallel evaluation of XQuery*. As explained in the Introduction, no comparable system is available yet. Therefore, for our comparison, we picked the BaseX centralized system (the best performing in the experiment above) and used Hadoop-MapReduce on one hand, and Stratosphere-PACT on the other hand, to parallelize its execution.

We compare PAXQuery, relying on the XML algebra-to-PACT translation we described, with the following alternative architecture. We deployed BaseX on each node, and parallelized XQuery execution based on these installed servers as follows:

1. Manually decompose each query into a set of leaf subqueries performing just tree pattern navigation, followed by a recomposition subquery which applies (possibly nested, outer) joins over the results of the leaf subqueries;
2. Parallelize the evaluation of the leaf subqueries through one Map over all the documents, followed by one Reduce to union all the results. Moreover, if the recomposition query is not empty, start a new MapReduce job running the recomposition XQuery query over all the results thus obtained, in order to compute complete query results.

This alternative architecture is in-between ChuQL [KCS11], where the query writer *explicitly* controls the functionality of the Map and Reduce blocks, i.e., MapReduce is *visible at the query level*, and PAXQuery where parallelism is completely hidden. In this architecture, q_1-q_8 translate to one `map` and one `reduce`, whereas q_9-q_{14} feature joins which translates into a recomposition query and thus a second job. As we will illustrate with the following example, the manual decomposition takes a considerable effort.

Example 1 (*continuation*). The MapReduce and PACT plans that execute the XQuery introduced in Section 4.2 are depicted in Figure 4.24.

Observe that the MapReduce workflow contains two jobs:

1. The first job creates a key/value pair out of each document in each collection, which contains the document's content. The pairs are correspondingly labeled so that they can be identified in the following steps. The `map` user function uses BaseX to execute a navigation query on the content of each input pair; q_1 is equivalent to the tree pattern e_1 in Figure 4.11, while q_2 is equivalent to e_2. In turn, the `reduce` gathers the pairs that originated from the same collection in a group, and applies a user function that unions the results for each of these collections, thus creating files f_1 and f_2 for collections 'people' and 'closed_auctions', respectively. Note that the `reduce` operation is necessary because we execute a nested outer join between the results from both collections

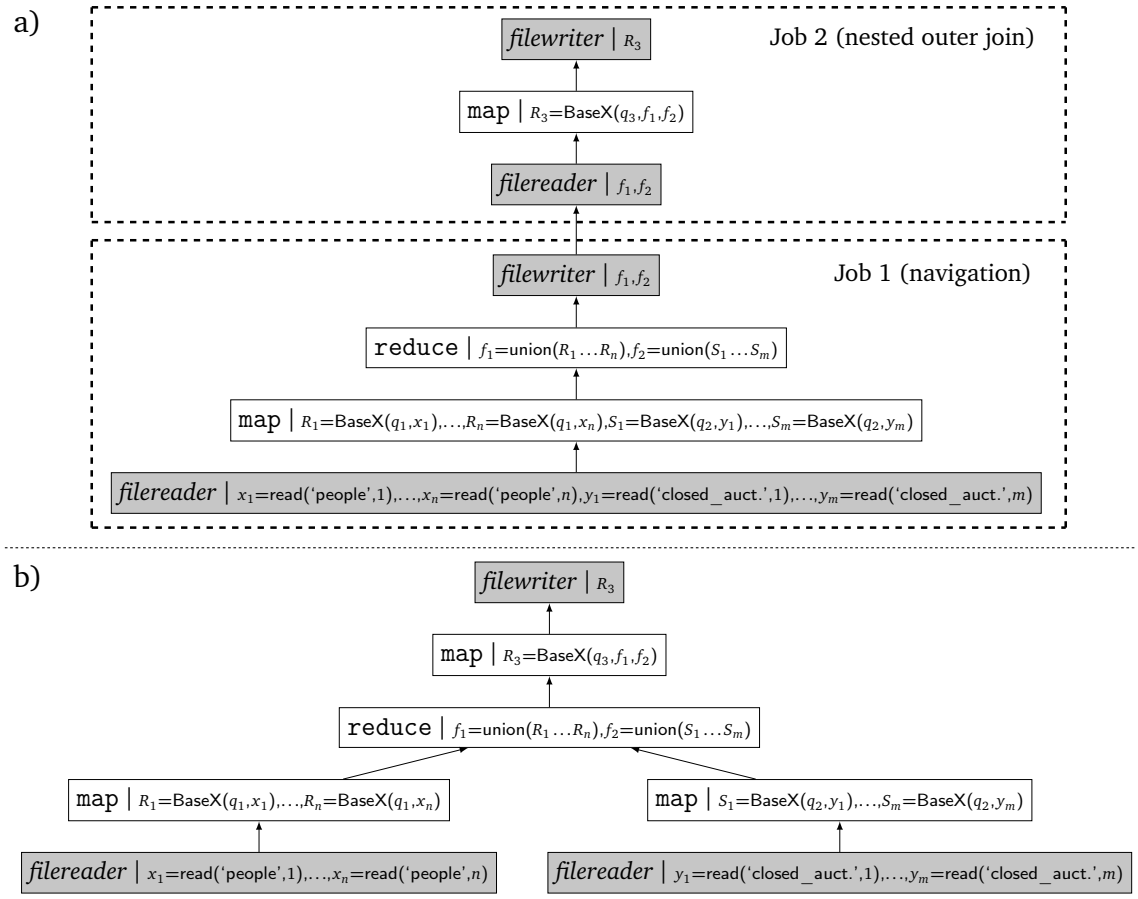


Figure 4.24: Execution of the XQuery in Example 1 using alternative architectures based on MapReduce (a) and PACT (b) for comparison with PAXQuery.

in the subsequent step. Without using our parallelization algorithms, it means that BaseX needs a global view over the results from each collection.

2. The second step is expressed as a Map-only job. It reads the inputs from the first job, and then it uses BaseX to execute the nested outer join between the inputs. The result is then written to disk.

The PACT plan is analog to the one that we just presented, but observe that in this case we do not have a linear workflow, but a DAG of operators. ◇

Table 4.5 shows the response times when running the query on the 8 nodes and 272GB; the shortest time is in bold. First, we notice that BaseX runs 2 to 5 times faster on Stratosphere than on Hadoop. This is due to Hadoop's checkpoints (writing intermediary results to disk) while Stratosphere currently does not perform such checkpoints, trading reliability for speed. For queries without joins (q_1-q_8), PAXQuery is faster for most queries than BaseX on Hadoop or Stratosphere; this simply points out that our in-house tree pattern matching operator (physical implementation of *nav*) is more efficient than the one of BaseX. Queries with joins (q_9-q_{14}) fail in the competitor architecture again. The reason is that intermediary join results grow too

Query	Evaluation time (seconds)		
	BaseX Hadoop-MR	**BaseX Stratosphere-PACT**	**PAXQuery**
q_1	465	**66**	70
q_2	773	282	**189**
q_3	762	243	**172**
q_4	244	72	**58**
q_5	237	72	**57**
q_6	488	**70**	73
q_7	245	74	**62**
q_8	576	237	**206**
q_9	~~OOM~~	~~OOM~~	**114**
q_{10}	~~OOM~~	~~OOM~~	**299**
q_{11}	~~OOM~~	~~OOM~~	**334**
q_{12}	~~OOM~~	~~OOM~~	**132**
q_{13}	~~OOM~~	~~OOM~~	**456**
q_{14}	~~OOM~~	~~OOM~~	**683**

Table 4.5: Query evaluation time (8 nodes, 272GB).

large and this leads to an out-of-memory error. PAXQuery evaluates such queries well, based on its massively parallel (outer) joins.

4.6.3 Conclusions of the experiments

Our experiments demonstrate the efficiency of an XQuery processor built on top of PACT.

First, our scalability evaluation has shown that the translation to PACT allows PAXQuery to parallelize every query execution step with no effort required to partition, redistribute data etc., and thus to scale out with the number of machines in a cluster. The only case where scale-up was not so good is q_{14} where we used a `cross` (cartesian product) to translate an inequality join; an orthogonal optimization here would be to use a smarter dedicated join operator for such predicates, e.g. [OR11].

Secondly, we have shown that PAXQuery outperforms competitor XQuery processors, whether centralized or distributed over Hadoop and Stratosphere. None of the competing processors was able to evaluate any of our queries with joins across documents on the data volumes we considered, highlighting the need for efficient parallel platforms for evaluating such queries.

4.7 Related work

Massively parallel XML query processing. In this area, MRQL [FLGP11] proposes a simple SQL-like XML query language implemented through a few operators directly compilable into MapReduce. Like our XQuery fragment, MRQL queries may be nested,

however, its dialect does not allow expressing the rich join flavours that we use. Further, the XML navigation supported by MRQL is limited to XPath, in contrast to our richer navigation based on tree patterns with multiple returning nodes, and nested and optional edges.

ChuQL [KCS11] is an XQuery extension that exposes the MapReduce framework to the developer in order to distribute computations among XQuery engines; this leaves the parallelization work to the programmer, in contrast with our implicitly parallel approach which does not expose the underlying parallelism at the query level.

HadoopXML [CLK+12] and the recent [BCM+13] process XML queries in Hadoop clusters by *explicitly fragmenting the input data* in a query-driven, respectively, schema-driven way, which is effective when querying one single huge document. In contrast, we focus on the frequent situation when no single document is too large for one node, but there are many documents whose global size is high, and queries may both *navigate and join* over them. Further, we do not require any partitioning work from the application level.

After the wide acceptance of Hadoop, other parallel execution engines and programming abstractions conceived to run custom data intensive tasks over large data sets have been proposed: PACT [BEH+10], Dryad [IBY+07], Hyracks [BCG+11] or Spark [ZCD+12]. Among these, the only effort at parallelizing XQuery is the ongoing VXQuery project [VXQ], translating XQuery into the Algebricks algebra, which compiles into parallel plans executable by Hyracks. In contrast, PAXQuery translates into an implicit parallel logical model such as PACT. Thus, our algorithms do not need to address underlying parallelization issues such as data redistribution between computation steps etc. which [BCG+11] explicitly mentions.

XQuery processing in centralized settings has been thoroughly studied, in particular through algebras in [RSF06, DPX04, MHM06, MPV09]. Our focus is on *extending the benefits of implicit large-scale parallelism to a complex XML algebra*, by formalizing its translation into the implicitly parallel PACT paradigm.

XML data management has also been studied from many other angles, e.g., on top of column stores [BGvK+06], distributed with [KOD10] or without [ABC+03] an explicit fragmentation specification, in P2P [KP05] etc. We focus on XQuery evaluation through the massively parallel PACT framework, which leads to specific translation difficulties we addressed.

Parallelizable nested languages. Recently, many high-level languages which translate into massively parallel frameworks have been proposed; some of them work with nested data and/or feature nesting in the language, thus somehow resemble XQuery.

Jaql [BEG+11] is a scripting language tailored to JSON data, which translates into MapReduce; Meteor [HRL+12], also for JSON, translates into PACT. None of these languages handles XQuery semantics exactly, since JSON does not feature node identity; the languages are also more limited, e.g., Jaql only supports equi-joins.

The Asterix Query Language [BBC+11], or AQL in short, is based on FLOWR expressions and resembles XQuery, but ignores node identity which is important in XQuery and which we support. Like VXQuery, AQL queries are translated into Algebricks; recall that unlike our translation, its compilation to the underlying Hyracks

engine needs to deal with parallelization related issues.

Finally, other higher level languages that support nested data models and translate into parallel processing paradigms include Pig Latin [ORS+08] or Hive [TSJ+10]. Our XQuery fragment is more expressive, in particular supporting more types of joins. In addition, Pig only allows two levels of nesting in queries, which is a limitation. In contrast, we translate XQuery into unnested algebraic plans with (possibly nested, possibly outer) joins and grouping which we parallelize, leading to efficient execution even for (originally) nested queries.

Complex operations using implicit parallel models. The problem of evaluating complex operations through implicit parallelism is of independent interest. For instance, the execution of join operations using MapReduce has been studied extensively. Shortly after the first formal proposal to compute equi-joins on MapReduce [YDHP07], other studies extending it [BPE+10, JTC11] or focusing on the processing of specific join types such as multi-way joins [AU10], set-similarity joins [VCL10], or θ-joins [OR11], appeared. PAXQuery is the first to translate a large family of joins (which can be used outside XQuery), into the more flexible PACT parallel framework.

4.8 Summary

This chapter has presented the PAXQuery approach for the implicit parallelization of XQuery, through the translation of an XQuery algebraic plan into a PACT parallel plan. We targeted a rich subset of XQuery 3.0 including recent additions such as explicit grouping, and demonstrated the efficiency and scalability of PAXQuery with experiments on collections of hundreds of GBs.

Acknowledgements This work has been partially funded by the KIC EIT ICT Labs activity 12115. We would like to thank Kostas Tzoumas for his valuable comments and suggestions on this work.

Chapter 5

Reuse-based Optimization for Pig Latin

This chapter presents a novel approach for identifying and reusing repeated subexpressions in Pig Latin scripts. In particular, we lay the foundation of our reuse-based algorithms by formalizing the semantics of the Pig Latin query language with extended nested relational algebra for bags. Our optimization algorithm, named *PigReuse*, operates on the algebraic representations of Pig Latin scripts. It identifies subexpression merging opportunities, selects the best ones to execute based on a cost function, and reuses their results as needed in order to compute exactly the same output as the original scripts. Our experiments demonstrate the efficiency and effectiveness of the PigReuse algorithm.

The material of this chapter is being considered for publication in an international conference.

5.1 Introduction

The efficient processing of very large volumes of data has lately relied on massively parallel processing models, of which MapReduce is the most widely adopted. However, the simplicity of the MapReduce model leads to relatively complex programs to express even moderately complex tasks. To facilitate the specification of data processing tasks to be executed in a massively parallel fashion, several higher-level query languages have been introduced, which are more user-friendly, and which are automatically compiled into MapReduce programs. Languages that have gained wide adoption include Pig Latin [ORS+08], HiveQL [TSJ+10], or Jaql [BEG+11].

In this work, we consider the Pig Latin language, which has raised significant interest from the application developers as well as the research community. Pig Latin provides dataflow-style primitives for expressing complex analytical data processing tasks. Pig Latin programs (also named *scripts*) are automatically optimized and compiled into MapReduce jobs by the Apache Pig system [Piga].

In a typical batch of Pig Latin scripts, there may be many identical (or equivalent)

sub-expressions, that is: script fragments applying the same processing on the same inputs, but appearing in distinct places within the same (or several) scripts. While the Pig Latin engine includes a query optimizer, it is currently not capable of recognizing such repeated subexpressions. As a consequence, they will be executed as many times as they appear in the Pig Latin script batch, whereas there is obviously an opportunity for enhancing performance by identifying common subexpressions, executing them only once, and reusing the results of the computation in every script needing them.

Identifying and reusing common subexpressions occurring in Pig Latin scripts automatically is the target of the present work. The problem bears obvious similarities with the known multi-query optimization and workflow reuse problems; however, as we discuss in Section 5.6, the Pig Latin setting leads to several novel aspects of the problem, which lead us to propose dedicated algorithms to solve them.

Motivating example. A Pig Latin script consists of a set of *binding expressions* and *store expressions*. Each binding expression follows the syntax var = op, meaning that the expression op will be evaluated, and the bag of tuples thus generated will be bound to the variable var. Then, var can be used by follow-up expressions in a script.

Consider the following Pig Latin script a_1:

```
1 A = LOAD 'page_views' AS (user, time, www);
2 B = LOAD 'users' AS (name, zip);
3 R = JOIN A BY user, B BY name;
4 S = FOREACH R GENERATE user, time, zip;
5 STORE S INTO 'a1out1';
6 T = JOIN A BY user LEFT, B BY name;
7 STORE T INTO 'a1out2';
```

Line 1 loads data from a file `page_views` and creates a bag of tuples that is bound to variable `A`. Each of these tuples consists of three attributes `(user,time,www)`. Line 2 loads data from a second file, and binds the resulting tuple bag to `B`. Line 3 joins the tuples of `A` and `B` based on the equality of the values bound to attributes `user` and `name`. The next line uses the important Pig Latin operator `FOREACH`, that applies a function on every tuple of the input bag. In this case, line 4 projects the attributes `user`, `time` and `zip` of every tuple in `C`. Then the result is stored in the file `a1out1`. In turn, line 6 executes a left outer join over the tuples of `A` and `B` based on the equality of the values bound to the same attributes `user` and `name`, and the result is stored in `a1out2`.

The following script a_2 only executes a left outer join over the same inputs:

```
1 A = LOAD 'page_views' AS (user, time, www);
2 B = LOAD 'users' AS (name, zip);
3 R = JOIN A BY user LEFT, B BY name;
4 STORE R INTO 'a2out';
```

The script b that we introduce next produces the same outputs as a_1 and a_2:

```
1 A = LOAD 'page_views' AS (user, time, www);
2 B = LOAD 'users' AS (name, zip);
3 R = COGROUP A BY user, B BY name;
4 S = FOREACH R GENERATE flatten(A), flatten(B);
5 T = FOREACH S GENERATE user, time, zip;
6 STORE T INTO 'a1out1';
7 U = FOREACH R GENERATE flatten(A),
8                  flatten (isEmpty(B) ? {(null,null,null)} : B);
9 STORE U INTO 'a1out2';
10 STORE U INTO 'a2out';
```

However, *b's execution time is* 45% *of the combined running time of* a_1 *and* a_2. The reason is twofold. First, observe that the joins are rewritten into a COGROUP [1] operation (line 3) and FOREACH operations (lines 4 and 7-8). The interest of cogroup is that through some simple restructuring, one can carve out of the cogroup output various flavors of joins (natural, outer, nested, semijoin etc.) This restructuring operation differs depending on whether we want to generate the join between A and B needed for script a_1 (line 4), or the left outer join between A and B for scripts a_1 and a_2 (lines 7-8). The detailed semantics of these restructuring operations will become clear in Section 5.4. Thus, the first reason for the speedup of b w.r.t. a_1 and a_2 is that the COGROUP output is reused to generate the result for both joins. The second reason is that in b, the left outer join is computed only once, and its result is used to produced the desired output of scripts a_1 (line 9) and a_2 (line 10).

Contributions. The technical contributions of this work are the following.

- We formalize the representation of Pig Latin scripts based on an existing well-established algebraic formalism, specifically Nested Relational Algebra for Bags (NRAB) [GM93]. This provides a formal foundation for accurately identifying common expressions in batches of Pig Latin scripts.
- We propose PigReuse, a multi-query optimization algorithm that merges equivalent subexpressions it identifies in Directed Acyclic Graph (DAGs) of NRAB operators corresponding to a batch of Pig Latin scripts. After identifying such reutilization opportunities, PigReuse produces an optimal merged plan where redundant computations have been eliminated. PigReuse relies on an efficient Binary Integer Linear Programming (BIP, in short) solver to select the best plan based on the cost function provided.
- We present extensions to our baseline PigReuse optimization algorithm to improve its *effectiveness*, i.e., increase the number of common subexpressions it detects.
- We have implemented PigReuse as an extension module within the Apache Pig system. We present an experimental evaluation of our techniques using two different cost functions to select the best plan.

Outline. Section 5.2 describes our approach to represent Pig Latin scripts as DAGs of NRAB operators. Section 5.3 presents PigReuse, our reuse-based query optimization approach focusing on identifying and merging common subexpressions. Section 5.4 details different strategies that we use to make our reuse-based optimization approach more effective. Section 5.5 describes our experimental evaluation. Finally, Section 5.6 discusses related work, and then we conclude.

1. COGROUP can be seen as a generalization of the *group-by* operation on two or more relations: for every value of the grouping key occurring in any of the inputs, it outputs a tuple that includes an attribute *group* bound to the grouping key, and a bag of tuples for each input R_i such that the bag R_i includes all tuples in R_i that contain the value of the grouping key.

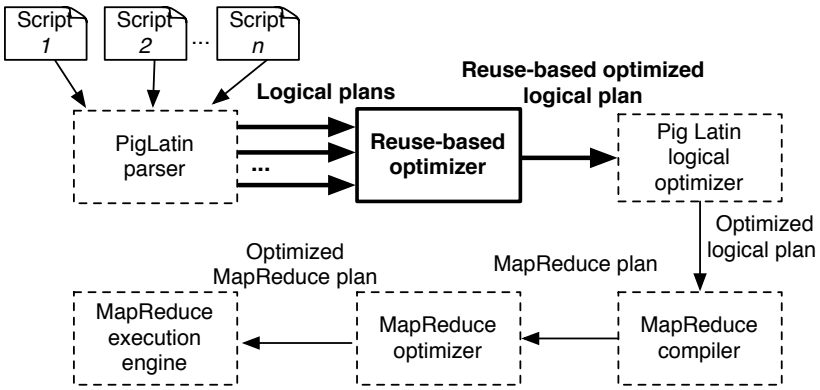


Figure 5.1: Integration of PigReuse optimizer within Pig Latin execution engine.

5.2 Algebraic representation of Pig Latin programs

Figure 5.1 depicts the integration of our reuse-based optimization into the Pig Latin architecture; modules, denoted by dashed lines, belong to the original Pig Latin query processor. As shown in the figure, our reuse-based optimizer works on the algebraic representation of Pig Latin scripts. Thus, our proposal is orthogonal to the Pig Latin query evaluation and execution process. This allows our approach (i) to benefit from the Pig Latin optimizer, and (ii) to apply our optimization independently of the underlying Pig Latin query compilation and execution engines.

The algebraic formalization of Pig Latin is necessary, as it ensures the *correctness* of the manipulations involved in the detection of common subexpressions within batches of Pig Latin scripts, based on known expression equivalence results [CV93, CM94, LW97]. The Pig Latin data model features complex data types (e.g., tuple, map etc.) and nested relations with duplicates (bags). Earlier work [ADD+11] stated that Pig Latin scripts can be translated to Nested Relational Algebra with bag semantics, but to the best of our knowledge, no formalization of such translation has been proposed to date. In this section, we present our translation of PigLatin to an extension of the Nested Relational Algebra for Bags [GM93] (NRAB, for short) that includes operators needed to support Pig Latin semantics.

In the following, Section 5.2.1 provides background on the NRAB, while Section 5.2.2 presents our translation of Pig Latin operators into the algebra operators. Finally, Section 5.2.3 describes the DAG representation of the translated scripts.

5.2.1 Extended NRAB

We consider a subset of the NRAB algebra and extend it with other operators. Table 5.1 lists all basic operators of NRAB (top part) and the additional operators we introduce (bottom part). All additional operators but *scan* and *store* are redundant, i.e., they can be expressed using the basic operators. We decided to introduce additional operators for two main reasons: (i) allowing a one-to-one representation of Pig Latin scripts into the algebra, and (ii) giving our algorithm additional opportuni-

Notation	Name	Input arity	Output description
ϵ	Duplicate elimination	Unary	Distinct tuples from the input relation.
$map\langle\varphi\rangle$	Restructure	Unary	All the tuples in the input after applying a function φ.
$\sigma\langle p\rangle$	Selection	Unary	All the tuples in the input that satisfy the boolean predicate p.
$\uplus$	Additive union	n-ary, $n \geq 2$	Union of input relations, including duplicates.
$-$	Substraction	Binary	Difference between relations, including duplicates.
$\times$	Cartesian product	n-ary, $n \geq 2$	Cartesian product of input relations, including duplicates.
δ	Bag-destroy function	Unary	Unnests one level for the tuples in the input relation.

Notation	Name	Input arity	Output description
$scan\langle fileID\rangle$	Load	-	Reads a file and loads it as a relation.
$store\langle dir\rangle$	Store	Unary	Writes the contents of the tuples for an input relation to a file.
$\pi\langle a_1,\ldots,a_n\rangle$	Projection	Unary	Projects attributes $a_1,\ldots,a_n$ from the input tuples.
$cogroup\langle a_1,\ldots,a_n\rangle$	Cogroup	n-ary, $n \geq 1$	Groups tuples together from input relations based on the equality of their values for attributes $(a_1,\ldots,a_n)$.
$\bowtie\langle a_1,\ldots,a_n\rangle$	Join	n-ary, $n \geq 2$	Returns the combination of tuples from input relations based on the equality of their values for attributes $(a_1,\ldots,a_n)$.
$\mathbin{⟕}\langle a_1,a_2\rangle$	Left outer join	Binary	Returns the combination of tuples from input relations for which $a_1{=}a_2$, and the tuples in the left relation without a matching right tuple.
$\mathbin{⟖}\langle a_1,a_2\rangle$	Right outer join	Binary	Returns the combination of tuples from input relations for which $a_1{=}a_2$, and the tuples in the right relation without a matching left tuple.
$\mathbin{⟗}\langle a_1,a_2\rangle$	Full outer join	Binary	Returns the combination of tuples from input relations for which $a_1{=}a_2$, the tuples in the left relation without a matching right tuple, and the tuples in the right relation without a matching left tuple.
$mapconcat\langle\varphi\rangle$	Restructure and concatenate	Unary	Applies $map\langle\varphi\rangle$ and concatenates its result to the original tuple.
empty	Empty function	Unary	The boolean function *empty* returns true if and only if the input relation is empty.
sum, *max*, *min*, *count*	Aggregate functions	Unary	Returns the sum of integer values for an attribute field in an input relation, maximum integer value, minimum integer value of an attribute field in an input relation, and total number of tuples in an input relation.

Table 5.1: Basic NRAB operators (top) and proposed extension (bottom) to express Pig Latin semantics.

ties to detect common subexpressions by exploring different rewritings. For instance, any type of join can (also) be expressed by a combination of *cogroup*, *restructure*, and *bag destroy*. Using this alternative representation of a join, it becomes easier to match it with a subexpression involving a *cogroup*, than it is to search for common subexpressions in the plans we would generate using the standard NRAB operators only.

The formal semantics of NRAB data model and these operators is described the following. First, we recall the NRAB data model in Section 5.2.1.1, while we present the subset of its operators that we use to represent Pig Latin semantics in Section 5.2.1.2. Then, Section 5.2.1.3 extends NRAB with the Pig Latin operators, whose semantics are defined using the subset of NRAB operators that we introduce previously.

5.2.1.1 Data model

Let us assume the existence of a set of domain names $\widehat{D}_1,\ldots,\widehat{D}_n$ and an infinitive set of attributes $a_1, a_2, \ldots$. Further, the domain names are associated with domains $D_1,\ldots,D_n$. The elements of the domains can be of either atomic type or complex type. A type is associated with each instance of a domain. Formally, *types* and *values* are

defined as follows:

- If $\widehat{D}_i \in \widehat{D}$ is a domain name, then $\widehat{D}_i$ denotes the *domain type*. For each database relation R in domain $\widehat{D}$, the type of R is $\widehat{D}_i$.
- If $T_1, \ldots, T_n$ are types and $a_1, \ldots, a_n$ are distinct attribute names for tuples in a database relation R, then R=$\{\{[a_1 : T_1, \ldots, a_n : T_n]\}\}$ is a bag of tuples in which $[a_1 : T_1, \ldots, a_n : T_n]$ is a *tuple* type. If v_1, ..., v_n are values of types T_1, ..., T_n, respectively, then $[a_1 : v_1, \ldots, a_n : v_n]$ is value of the *tuple* type. We also include $T_{[]}$ as a type; the only value of this type is $[]$, the empty tuple.
- A *bag* is a (homogeneous) collection of tuples that may contain duplicates. If T is a tuple type, then $\{\{T\}\}$ is a *bag* type, whose domain is a set of bags containing homogeneous tuples of type T. We say that an element o *n-belongs* to a bag, if element o has n occurrences in that bag.
- A *bag database* is a set of named bags. A *bag schema* is an expression $B : T$, where B is a bag name and T is a bag type. An instance of B is a bag of type T.

5.2.1.2 Basic operators

NRAB operators. We now describe the NRAB operators [GM93] that we use to express Pig Latin sementics. The input and output types of all these operators are *bag* type.

- Duplicate elimination (ϵ). This operator extracts the distinct tuples in a relation. $\epsilon(R)$ is a bag containing exactly one occurrence of each tuple in R i.e., an element o *1-belongs* to $\epsilon(R)$ iff o *p-belongs* to R for some $p > 0$, and *0-belongs* to $\epsilon(R)$ otherwise.
- Restructuring (*map*). $map\langle\varphi\rangle(R)$ returns a bag of type $\{\{T\}\}$, constructed by applying a function φ on each element of R. This operation is introduced for performing restructuring of complex values, which may include the application of functions to substructures of the values. *map* is a higher order operation with a function parameter φ that describes the restructuring.
- Selection (σ). Given a bag R and a boolean valued predicate condition p, $\sigma\langle p\rangle(R)$ denotes the select operation that returns a bag containing all the elements of R that satisfy the condition p. Only unary predicates can be used as parameters for the select; we refer to them as *select specifications*.
- Additive union ($\uplus$). This operator deals with the union of bags with possibly duplicate elements. If R and S are two input relations of *bag* type $\{\{T\}\}$, then $R \uplus S$ is a bag of type $\{\{T\}\}$, such that a tuple t of type T *n-belongs* to $R \uplus S$, iff t *p-belongs* to R and *q-belongs* to S and $n = p + q$.
- Substraction ($-$). If R and S are two input relations of *bag* type $\{\{T\}\}$, then $R-S$ is a bag of type $\{\{T\}\}$, such that a tuple t of type T *n-belongs* to $R-S$, iff t *p-belongs* to R, *q-belongs* to S and $n = max(0, p-q)$, where function *max* returns the highest among the input values 0 and $p - q$.
- Cartesian product ($\times$). If R and S are bags containing tuples of arity k and k' respectively, then $R \times S$ is a bag containing tuples of arity $k + k'$, such that the new relation X becomes, $X = R \times S = \{[a_1, \ldots, a_k, a_{k+1}, \ldots, a_{k+k'}]\}$, where

$[a_1,\ldots,a_k,a_{k+1},\ldots,a_{k+k'}]$ is a tuple type. Tuple $t=[a_1,\ldots,a_k,\ a_{k+1},\ldots,a_{k+k'}]$ *n-belongs* to $R\times S$ iff $t_1=[a_1,\ldots,a_k]$ *p-belongs* to R and $t_2=[a_{k+1},\ldots,a_{k+k}]$ *q-belongs* to S and $n=pq$.

– Bag-destroy function (δ). δ unnests one level of bag nesting. If R is a bag of type $\{\{S:\{\{T\}\}\}\}$, then $map\langle\delta(S)\rangle(R)$ results a bag of type {{T}}.

NRAB functions. Function definition in NRAB has two parts: a class of base functions and function constructors that are used for constructing more complex function expressions.

First, we describe the base functions. In our algebra, constants c, and database relation names $\widehat{R}$ are considered as functions. Additionally, each attribute of the input relation is also considered as a function expression. We use **id** for denoting the identity function. For example, $map\langle R\uplus\mathbf{id}\rangle(S)$, denotes that additive union of R's element is performed recursively on each of S's elements, where S is a bag of tuples. Here, **id** indicates each element in S. The algebraic operations, except *select* and *restructuring*, are function expressions. *Select* and *restructuring* are function constructors, which are discussed next.

In our algebra, complex functions are constructed by using one of the function construction operators (*select* and *restructuring*). If φ is a unary function, then $map\langle\varphi\rangle(R)$ is a function. Similarly, if p is a unary boolean-valued function then $\sigma\langle p\rangle(R)$ is also a function. We use tuple construction as a function constructor i.e., if $f_1,\ldots,f_n$ are unary functions, then $[f_1,\ldots,f_n]$ is a unary function, whose meaning is defined by $[f_1,\ldots,f_n](x)=[f_1(x),\ldots,f_n(x)]$. Our algebra supports labeled tuple construction as a function constructor too, i.e., formation of expressions like $[A_1=f_1,\ldots,A_n=f_n]$ is allowed; note that the A_is here are not functions but labels. The semantics is given by $[A_1=f_1,\ldots,A_n=f_n](x)=[A_1:f_1(x),\ldots,A_n:f_n(x)]$. This implies that every function is unary, where its input is a tuple.

5.2.1.3 Additional operators

In the following, we extend the basic NRAB set of operators to encapsulate the semantics of more complex operations that are supported by the Pig Latin language.

– Scan (*scan*). $scan\langle fileID\rangle$ is an operator introduced to represent a data source that reads a file *fileID*.
– Store (*store*). $store\langle dir\rangle(R)$ is an operator introduced to represent a data sink that writes the bag R to directory *dir*.
– Projection (π). $\pi\langle a_1,\ldots,a_n\rangle(R)$ projects attributes with names $a_1,\ldots,a_n$ from the tuples in bag R. Formally:

$$\pi\langle a_1,\ldots,a_n\rangle(R)\equiv map\langle[a_1,\ldots,a_n]\rangle(R)$$

– Cogroup (*cogroup*). In order to define the semantics of the *cogroup* operator, we first define a G operator that works on a single bag. In particular, $G\langle a\rangle(R)$ groups the tuples in R by the value bound to a. The result of the expression is a bag with tuples containing two elements: a *group* attribute associated to the grouping value, and a R attribute associated to the bag of tuples whose attribute

a was bound to that value. Formally:

$$G\langle a\rangle(R) \equiv map\langle map\langle \sigma\langle group{=}a\rangle(id)\rangle(R)\rangle\ (map\langle [group = a, R = R]\rangle(R))$$

$cogroup\langle a_1,\ldots,a_n\rangle(R_1,\ldots,R_n)$ groups together tuples from multiple bags $R_1,\ldots,R_n$, based on the values of their attributes $a_1,\ldots,a_n$, respectively. The result of a *cogroup* operation is a bag containing a $group$ attribute, bound to values of attributes $a_1,\ldots,a_n$, followed by one bag of grouped tuples for each relation in $R_1,\ldots,R_n$. Without loss of generality, we define it formally for two input relations; the extension for more than two inputs is straightforward. Thus:

$$cogroup\langle a_1, a_2\rangle(R_1,R_2) \equiv A_9$$

where:

$$\mathscr{A}_1 := G\langle a_1\rangle(R_1) \qquad \mathscr{A}_2 := G\langle a_2\rangle(R_2)$$
$$\mathscr{A}_3 := \bowtie\langle group{=}group\rangle(\mathscr{A}_1,\mathscr{A}_2) \quad \mathscr{A}_4 := \pi\langle group\rangle(\mathscr{A}_3)$$
$$\mathscr{A}_5 := \pi\langle group\rangle(\mathscr{A}_1)\ \mathscr{A}_6 := \bowtie\langle group{=}group\rangle(\mathscr{A}_5{-}\mathscr{A}_4,\mathscr{A}_1)$$
$$\mathscr{A}_7 := \pi\langle group\rangle(\mathscr{A}_2)\ \mathscr{A}_8 := \bowtie\langle group{=}group\rangle(\mathscr{A}_7{-}\mathscr{A}_4,\mathscr{A}_2)$$
$$\mathscr{A}_9 := \mathscr{A}_3 \uplus \mathscr{A}_6 \uplus \mathscr{A}_8$$

– Inner join ($\bowtie$). $\bowtie\langle a_1, a_2,\ldots,a_n\rangle(R_1,R_2,\ldots,R_n)$ creates the cartesian product between the tuples in bags $R_1,R_2,\ldots,R_n$, and filters the resulting tuples based on condition $a_1{=}a_2{=}\ldots{=}a_n$. Thus, $\bowtie$ is formalized as:

$$\bowtie\langle a_1, a_2,\ldots,a_n\rangle(R_1,R_2,\ldots,R_n) \equiv \sigma\langle a_1{=}a_2{=}\ldots{=}a_n\rangle(R_1 \times R_2 \times \ldots \times R_n)$$

– Left outer join ($\overrightarrow{\bowtie}$). $\overrightarrow{\bowtie}\ \langle a_1{=}a_2\rangle(R_1,R_2)$ returns the cartesian product of tuples from input relations R_1 and R_2 for which boolean condition $a_1{=}a_2$ is true, and the tuples in R_1 without a matching right tuple. Formally:

$$\overrightarrow{\bowtie}\ \langle a_1, a_2\rangle(R_1,R_2) \equiv A_5$$

where:

$$\mathscr{A}_1 := \bowtie\langle a_1, a_2\rangle(R_1,R_2) \qquad \mathscr{A}_2 := \pi\langle a_1\rangle(\mathscr{A}_1)$$
$$\mathscr{A}_3 := \pi\langle a_1\rangle(R_1) \quad \mathscr{A}_4 := \bowtie\langle a_1, a_1\rangle(\mathscr{A}_3{-}\mathscr{A}_2,\mathscr{A}_1)$$
$$\mathscr{A}_5 := \mathscr{A}_1 \uplus \mathscr{A}_4$$

– Right outer join ($\overleftarrow{\bowtie}$). $\overleftarrow{\bowtie}\ \langle a_1{=}a_2\rangle(R_1,R_2)$ returns the cartesian product of tuples from input relations R_1 and R_2 for which boolean condition $a_1{=}a_2$ is true, and the tuples in R_2 without a matching right tuple. Formally:

$$\overleftarrow{\bowtie}\ \langle a_1, a_2\rangle(R_1,R_2) \equiv A_5$$

where:

$$\mathscr{A}_1 := \bowtie\langle a_1, a_2\rangle(R_1,R_2) \qquad \mathscr{A}_2 := \pi\langle a_2\rangle(\mathscr{A}_1)$$
$$\mathscr{A}_3 := \pi\langle a_2\rangle(R_2) \quad \mathscr{A}_4 := \bowtie\langle a_2, a_2\rangle(\mathscr{A}_3{-}\mathscr{A}_2,\mathscr{A}_1)$$
$$\mathscr{A}_5 := \mathscr{A}_1 \uplus \mathscr{A}_4$$

– Full outer join ($\overleftrightarrow{\bowtie}$). $\overleftrightarrow{\bowtie}\ \langle a_1{=}a_2\rangle(R_1,R_2)$ returns the cartesian product of tuples from input relations R_1 and R_2 for which boolean condition $a_1{=}a_2$ is true, the tuples in R_1 without a matching right tuple, and the tuples in R_2 without a matching left tuple. Formally:

$$⟗ \langle a_1, a_2\rangle(R_1,R_2) \equiv A_8$$

where:

$$\mathcal{A}_1 := \bowtie\langle a_1,a_2\rangle(R_1,R_2)$$
$$\mathcal{A}_2 := \pi\langle a_1\rangle(\mathcal{A}_1) \quad \mathcal{A}_3 := \pi\langle a_1\rangle(R_1) \; \mathcal{A}_4 := \bowtie\langle a_1,a_1\rangle(\mathcal{A}_3 - \mathcal{A}_2, \mathcal{A}_1)$$
$$\mathcal{A}_5 := \pi\langle a_2\rangle(\mathcal{A}_1) \quad \mathcal{A}_6 := \pi\langle a_2\rangle(R_2) \; \mathcal{A}_7 := \bowtie\langle a_2,a_2\rangle(\mathcal{A}_6 - \mathcal{A}_5, \mathcal{A}_1)$$
$$\mathcal{A}_8 := \mathcal{A}_1 \uplus \mathcal{A}_4 \uplus \mathcal{A}_7$$

– Restructuring and concatenation (*mapconcat*). The operation $mapconcat\langle\varphi\rangle(R)$ applies $map\langle\varphi\rangle(R)$ and concatenates its result to the original tuple. Thus:

$$mapconcat\langle\varphi\rangle(R) \equiv map\langle[id, \varphi]\rangle(R)$$

– Empty (*empty*) and aggregate functions (*aggr*). The boolean function $empty(R)$ returns true iff R is empty. In turn, aggregate functions *aggr* include *count*, *max*, *min* and *sum*. $count(R)$ calculates the number elements in a bag of tuples R. $max\langle a\rangle(R)$ returns the maximum integer value of an element a in a bag of tuples R. $min\langle a\rangle(R)$ returns the minimum integer value of an element a in a bag of tuples R. $sum\langle a\rangle(R)$ returns the sum of integer values for an element a in a bag of tuples R. Each of these functions can be described in NRAB. For the sake of presentation, we do not further describe the semantics of these functions in this thesis.

5.2.2 Pig Latin translation

Along the lines of [RSF06], we define our Pig Latin to NRAB translation by means of deduction (or *translation*) *rules*. In a nutshell, a rule describes *how the translation is performed when some conditions are met over the input*. Our rules rely on *translation judgments,* noted as J, J_i, and are of the form:

$$\frac{J_1 \;\ldots\; J_n}{J}$$

stating that the translation J (conclusion) is recursively made in terms of translations $J_1 \;\ldots\; J_n$ (premises). The translation judgments J_i are optional.

For ease of presentation, we split the rules in two sets: the first one deals with the translation of programs as ordered sequences of expressions, while the second set details the translation of a single Pig Latin operation. Below, we present the rule sets in turn.

Pig Latin scripts translation. Rules in the first set rely on judgments of the form:

$$[\![P]\!]_\Gamma \rightsquigarrow \Gamma'$$

which reads as:

“ A Pig Latin program P is translated to a set of named NRAB expressions Γ', in the context of a given set of named NRAB expressions Γ.”

$$\frac{[\![\mathsf{expr}_1]\!]_{\Gamma_0} \rightsquigarrow \Gamma_1 \quad \cdots \quad [\![\mathsf{expr}_n]\!]_{\Gamma_{n-1}} \rightsquigarrow \Gamma_n}{[\![\mathsf{expr}_1;\ldots;\mathsf{expr}_n;]\!]_{\Gamma_0} \rightsquigarrow \Gamma_n} \quad \text{(SCRIPT)}$$

$$\frac{\mathsf{op} \Rightarrow \mathscr{A} \qquad \Gamma_1 := \Gamma_0 \cup \{var = \mathscr{A}\}}{[\![\underline{\mathsf{var}} = \mathsf{op}]\!]_{\Gamma_0} \rightsquigarrow \Gamma_1} \quad \text{(BIND)}$$

$$\frac{\mathscr{A} := store\langle dir\rangle(var) \qquad \Gamma_1 := \Gamma_0 \cup \{\top = \mathscr{A}\}}{[\![\texttt{STORE}\ \underline{\mathsf{var}}\ \texttt{INTO}\ \underline{\mathsf{dir}}]\!]_{\Gamma_0} \rightsquigarrow \Gamma_1} \quad \text{(STORE)}$$

Figure 5.2: Translation rules for Pig Latin scripts and basic Pig Latin constructs.

By rules definition, it easily follows that Γ' always includes Γ.

A named NRAB expression is a binding of the form $\{\underline{\mathsf{var}} = \mathscr{A}\}$ where $\underline{\mathsf{var}}$ is a name given to the algebraic expression $\mathscr{A}$. During the application of the translation rules, every binding expression $\{\underline{\mathsf{var}} = \mathsf{op}\}$ belonging to the Pig Latin program is translated into a named algebraic expression $\{var = \mathscr{A}\}$, where $\mathscr{A}$ is the NRAB expression corresponding to the operation op (and obtained by applying the second set of translation rules).

Binding expressions in the Pig Latin program are translated one after the other, according to their order in the program. Each time a named algebraic expression $\{var = \mathscr{A}\}$ is created, it is added to the context Γ. The context holds all variables which may be encountered while translating subsequent Pig Latin binding expressions of the program; we assume that *var* is a fresh variable, i.e., it is not already bound in the context.

Figure 5.2 shows the rules used by the high-level translation process outlined above. The rules are rather simple; note that the rule corresponding to `STORE` adds to the context a dummy binding. This rule records the fact that a bag has been saved on the disk, thus the symbol $\top$ is used instead of a variable symbol, which is not needed in this case.

Pig Latin operations translation. The second set of rules translates the operator op from a binding expression $\underline{\mathsf{var}} = \mathsf{op}$ into a NRAB expression $\mathscr{A}$. These rules are defined over judgements of the form:

$$\mathsf{op} \Rightarrow \mathscr{A}$$

which reads as:

"A Pig Latin operation op is translated to a NRAB expression $\mathscr{A}$."

Most Pig Latin operators have a one-to-one correspondence with NRAB operators, hence the related translation is straightforward. Figure 5.3 shows the translation rules for these Pig Latin operators. In the following we describe each of these rules.

Rule (LOAD) translates a `LOAD` expression into a *scan* that generating a new bag that satisfies the schema description in the input expression.

$$\frac{\mathscr{A} := scan\langle fileID\rangle}{\texttt{LOAD}\ fileID \Rightarrow \mathscr{A}} \quad \text{(LOAD)}$$

$$\frac{\mathscr{A} := \epsilon(\underline{\text{var}}_1)}{\texttt{DISTINCT}\ \underline{\text{var}}_1 \Rightarrow \mathscr{A}} \quad \text{(DISTINCT)}$$

$$\frac{\mathscr{A} := \sigma\langle p\rangle(\underline{\text{var}}_1)}{\texttt{FILTER}\ \underline{\text{var}}_1\ \texttt{BY}\ p \Rightarrow \mathscr{A}} \quad \text{(FILTER)}$$

$$\frac{\mathscr{A} := \delta(\underline{\text{var}})}{\texttt{FLATTEN}(\underline{\text{var}}) \Rightarrow \mathscr{A}} \quad \text{(FLATTEN FUNCTION)}$$

$$\frac{\mathscr{A} := empty(\underline{\text{var}})}{\texttt{IsEmpty}(\underline{\text{var}}) \Rightarrow \mathscr{A}} \quad \text{(EMPTY FUNCTION)}$$

$$\frac{\mathscr{A} := aggr(\underline{\text{var}})}{\texttt{AGGR}(\underline{\text{var}}) \Rightarrow \mathscr{A}} \quad \text{(AGGREGATION FUNCTION)}$$

$$\frac{\mathscr{A} := \underline{\text{var}}_1 \uplus \ldots \uplus \underline{\text{var}}_n}{\texttt{UNION}\ \underline{\text{var}}_1, \ldots, \underline{\text{var}}_n \Rightarrow \mathscr{A}} \quad \text{(UNION)}$$

$$\frac{\mathscr{A} := \underline{\text{var}}_1 \times \ldots \times \underline{\text{var}}_n}{\texttt{CROSS}\ \underline{\text{var}}_1, \ldots, \underline{\text{var}}_n \Rightarrow \mathscr{A}} \quad \text{(CROSS)}$$

$$\frac{\mathscr{A} := cogroup\langle a_1, \ldots, a_n\rangle(\underline{\text{var}}_1, \ldots, \underline{\text{var}}_n)}{\texttt{COGROUP}\ \underline{\text{var}}_1\ \texttt{BY}\ a_1, \ldots, \underline{\text{var}}_n\ \texttt{BY}\ a_n \Rightarrow \mathscr{A}} \quad \text{(GOGROUP)}$$

$$\frac{\mathscr{A}_1 := \bowtie\langle a_1, \ldots, a_n\rangle(\underline{\text{var}}_1, \ldots, \underline{\text{var}}_n)}{\texttt{JOIN}\ \underline{\text{var}}_1\ \texttt{BY}\ a_1, \ldots, \underline{\text{var}}_n\ \texttt{BY}\ a_n \Rightarrow \mathscr{A}_1} \quad \text{(INNER JOIN)}$$

$$\frac{\mathscr{A}_1 := ⟕\langle a_1, a_2\rangle(\underline{\text{var}}_1, \underline{\text{var}}_2)}{\texttt{JOIN}\ \underline{\text{var}}_1\ \texttt{BY}\ a_1\ \texttt{LEFT}, \underline{\text{var}}_2\ \texttt{BY}\ a_2 \Rightarrow \mathscr{A}_1} \quad \text{(LEFT OUTER JOIN)}$$

$$\frac{\mathscr{A}_1 := ⟖\langle a_1, a_2\rangle(\underline{\text{var}}_1, \underline{\text{var}}_2)}{\texttt{JOIN}\ \underline{\text{var}}_1\ \texttt{BY}\ a_1\ \texttt{RIGHT}, \underline{\text{var}}_2\ \texttt{BY}\ a_2 \Rightarrow \mathscr{A}_1} \quad \text{(RIGHT OUTER JOIN)}$$

$$\frac{\mathscr{A}_1 := ⟗\langle a_1, a_2\rangle(\underline{\text{var}}_1, \underline{\text{var}}_2)}{\texttt{JOIN}\ \underline{\text{var}}_1\ \texttt{BY}\ a_1\ \texttt{FULL}, \underline{\text{var}}_2\ \texttt{BY}\ a_2 \Rightarrow \mathscr{A}_1} \quad \text{(FULL OUTER JOIN)}$$

Figure 5.3: Rules for translating Pig Latin operators to corresponding NRAB representations.

Rule (DISTINCT) translates `DISTINCT` into a ϵ operator on the input relation $\underline{\text{var}}_1$.

Rule (FILTER) translates a Pig Latin `FILTER` operator into a selection σ with a condition p on $\underline{\text{var}}_1$.

Rule (FLATTEN FUNCTION) translates `FLATTEN` into a δ function that unnests the bag $\underline{\text{var}}$.

$$\frac{}{\texttt{FOREACH}\ \underline{\mathsf{var}}\ \texttt{GENERATE}\ \underline{\mathsf{var}}_1\ \ldots, \underline{\mathsf{var}}_n \Rightarrow \pi\langle \underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n\rangle(\underline{\mathsf{var}})} \quad \text{(PROJECTION FE)}$$

$$\frac{\underline{\mathsf{f}}_1 \Rightarrow \mathscr{A}'_1, \ldots, \underline{\mathsf{f}}_m \Rightarrow \mathscr{A}'_m \qquad \mathscr{A}_2 := map\langle[\mathscr{A}'_1, \ldots, \mathscr{A}'_m]\rangle(var_1)}{\texttt{FOREACH}\ \underline{\mathsf{var}}\ \texttt{GENERATE}\ \underline{\mathsf{f}}_1, \ldots, \underline{\mathsf{f}}_m \Rightarrow \mathscr{A}_2} \quad \text{(SIMPLE FE)}$$

$$\frac{\begin{array}{c}\mathsf{op}_1 \Rightarrow \mathscr{A}'_1 \qquad \mathscr{A}_1 := mapconcat\langle[nvar_1 = \mathscr{A}'_1]\rangle(var_1) \\ \mathsf{op}_i \Rightarrow \mathscr{A}'_i \qquad \mathscr{A}_i := mapconcat\langle[nvar_i = \mathscr{A}'_i]\rangle(\mathscr{A}_{i-1}) \qquad 2 \leq i \leq n \\ \underline{\mathsf{f}}_1 \Rightarrow \mathscr{A}''_1, \ldots, \underline{\mathsf{f}}_m \Rightarrow \mathscr{A}''_m \qquad \mathscr{A}_{n+1} := map\langle[\mathscr{A}''_1, \ldots, \mathscr{A}''_m]\rangle(\mathscr{A}_n)\end{array}}{\texttt{FOREACH}\ \underline{\mathsf{var}}_1\ \{\underline{\mathsf{nvar}}_1 = \mathsf{op}_1; \ldots; \underline{\mathsf{nvar}}_n = \mathsf{op}_n;\ \texttt{GENERATE}\ \underline{\mathsf{f}}_1, \ldots, \underline{\mathsf{f}}_m\} \Rightarrow \mathscr{A}_{n+1}} \quad \text{(COMPLEX FE)}$$

Figure 5.4: Translation rules for `foreach` operator.

Rule (AGGREGATION FUNCTION) translates Pig Latin aggregation functions into the NRAB aggregate operators counterparts.

The functions introduced in the last two rules are blocks that need to be used in the algebra in conjunction with an *map* operator.

Rule (CROSS) translates a `CROSS` into a cartesian product between $\underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n$.

Rule (GOGROUP) translates a Pig Latin `COGROUP` operation to its algebraic equivalence *cogroup* that groups the tuples in $\underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n$ based on the values of attributes bound to $a_1, \ldots, a_n$.

Rule (INNER JOIN) translates an inner join `JOIN` operator into its algebraic counterpart $\bowtie$. Rule (LEFT OUTER JOIN) translates a Pig Latin left outer join expression into a $\mathbin{⟕}$ operator, while rule (RIGHT OUTER JOIN) translates a Pig Latin right outer join expression into a $\mathbin{⟖}$ operator. Finally, rule (FULL OUTER JOIN) translates a Pig Latin full outer join expression into a $\mathbin{⟗}$ operator. Observe that outer joins can only be binary in Pig Latin.

A special case is the `FOREACH` operator, whose translation is not trivial as it is the main way to write complex programs in Pig Latin, e.g., it allows applying nested operations. The translation rules for this operator are shown in Figure 5.4. We use three different rules depending on the form of the `FOREACH` expression:

- The first rule (PROJECTION FE) deals with the case of an iteration simply projecting n fields of the input relation. The rule specific to this case enables the generation of NRAB projections, playing an important role in our optimization technique. In Figure 5.4, $\underline{\mathsf{var}}_1$,$\underline{\mathsf{var}}_2$,...,$\underline{\mathsf{var}}_n$ are the fields to be projected from the input relation denoted by the name $\underline{\mathsf{var}}$.
- If the previous rule does not apply, and if the `FOREACH` operator contains a `GENERATE` clause with functions applied on the input relation $\underline{\mathsf{var}}$, the second rule (SIMPLE FE) is applied. In this rule, every function definition $\underline{\mathsf{f}}_i$ inside the `GENERATE` clause is translated to an algebraic expression $\mathscr{A}'_i$ and these expressions are applied with a *map* operator on each tuple in $\underline{\mathsf{var}}_1$ (recall Table 5.1).
- Rule (COMPLEX FE) in Figure 5.4 considers `FOREACH` expressions containing one or more binding expressions before the `GENERATE` clause. Each Pig Latin operator op_i is translated first into an algebraic expression $\mathscr{A}'_i$. These algebraic

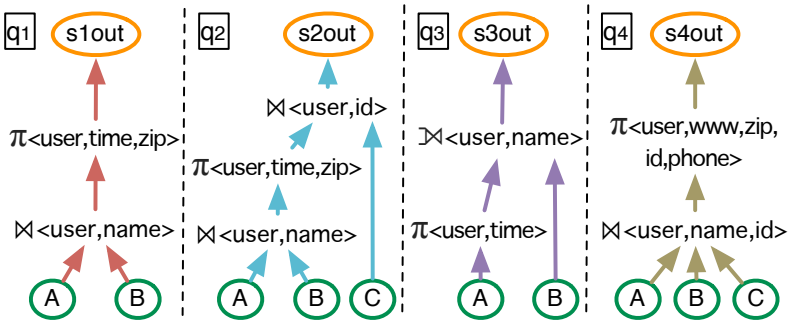
s_1
```
A = LOAD 'page_views' AS (user, time, www);
B = LOAD 'users' AS (name, zip);
R = JOIN A BY user, B BY name;
S = FOREACH R GENERATE user, time, zip;
STORE S INTO 's1out';
```

s_2
```
A = LOAD 'page_views' AS (user, time, www);
B = LOAD 'users' AS (name, zip);
C = LOAD 'power_users' AS (id, phone);
R = JOIN A by user, B BY name;
S = FOREACH R GENERATE user, time, zip;
T = JOIN S BY user, C by id;
STORE T INTO 's2out';
```

s_3
```
A = LOAD 'page_views' AS (user, time, www);
B = LOAD 'users' AS (name, zip);
R = FOREACH A GENERATE user, time;
S = JOIN R by user LEFT, B by name;
STORE S INTO 's3out';
```

s_4
```
A = LOAD 'page_views' AS (user, time, www);
B = LOAD 'users' AS (name, zip);
C = LOAD 'power_users' AS (id, phone);
R = JOIN A BY user, B by name, C by id;
S = FOREACH R GENERATE user, www, zip, id, phone;
STORE S INTO 's4out';
```

(a)



(b)

Figure 5.5: Sample Pig Latin scripts (a) and their corresponding algebraic DAG representation (b).

expressions are then used by a *mapconcat* operator, which applies $\mathscr{A}'_i$ on each tuple in $\mathscr{A}'_{i-1}$ (or $\underline{var}_1$ initially) and appends the result to the input tuple; the use of *mapconcat* is necessary to use local contextual information that is visible only in the scope of the translated `FOREACH` expression. Every function definition f_i inside the `GENERATE` clause is then translated to an algebraic expression $\mathscr{A}''_i$, which are applied on each of the resulting tuples from the algebraic expression $\mathscr{A}_n$.

5.2.3 DAG-structured NRAB queries

Let P be a Pig Latin program, and Γ be a set of NRAB binding expressions obtained from P via the translation process described above. We define below the DAG representation of Γ as follows.

Definition 14. *Given a context (set of bindings)* $\Gamma = \{var_1{=}\mathscr{A}_1, \ldots, var_n{=}\mathscr{A}_n\}$*, its DAG*

representation is a pair $(V, \vec{E})$*. V is a set of* $\langle var_i, op_i^{\mathscr{A}} \rangle$ *tuples such as for each* $v_i \in V$*:*

- var_i *is the variable associated to the node (thus, it is the unique identifier of a node);*
- $op_i^{\mathscr{A}}$ *is the top-most algebraic operator in the expression bound to* var_i*;*

Further, $\vec{E}$ *is a set of edges representing the data flow among the nodes of V. Specifically, there is an edge* $e_{i,j} \in \vec{E}$ *from* v_i *to* v_j*, iff the operation* $op_j^{\mathscr{A}}$ *is applied on the bag of tuples produced by* $op_i^{\mathscr{A}}$*.* ◇

In our DAG representation, a *source* i.e., a node with no incoming edges, always contains a *scan* operator. In turn, a *sink* i.e., a node with no outgoing edges, always corresponds to a *store* operator.

For illustration, Figure 5.5.a introduces four different Pig Latin scripts s_1-s_4, while their corresponding algebraic representation is shown in Figure 5.5.b. We will reuse these sample scripts throughout this chapter. The scripts read data from the three input relations `page_views`, `users`, and `power_users`; from now on, we depict these relations as `A`, `B`, and `C` in the algebraic plans, and we refer to them in the same fashion.

To illustrate, consider the script s_1, whose translation yields:

$$\begin{aligned}\Gamma = \{\, A &= scan\langle \text{`}page\_views\text{'}\rangle,\\ B &= scan\langle \text{`}users\text{'}\rangle,\\ R &= \bowtie \langle user, name\rangle(A, B),\\ S &= \pi\langle user, time, zip\rangle(R),\\ &store\langle \text{`}s1out\text{'}\rangle(S)\ \}\end{aligned}$$

After connecting the different algebraic expressions, we obtain the DAG query q_1 shown in Figure 5.5.b.

5.3 Reuse-based optimization

We have previously shown how to translate Pig Latin scripts into NRAB DAGs. Based on this DAG formalism, we now introduce our PigReuse algorithm that optimizes the query plans corresponding to a batch of scripts by reusing results of repeated subexpressions.

More specifically, given a collection of NRAB DAG queries Q, PigReuse proceeds in two steps:

Step (1). *Identify and merge all the equivalent subexpressions in Q.* To this end, we use an AND-OR DAG, in which an AND-node (or operator node) corresponds to an algebraic operation in Q, while an OR-node (or equivalence node) represents a set of subexpressions that generate the same result bag.

Step (2). *Find the optimal plan from the AND-OR DAG.* Based on a cost model, we make a globally optimal choice of the set of operator nodes to be actually evaluated. Our approach is independent of the particular cost function chosen; we discuss in Section 5.5.2 the functions that we have implemented for PigReuse.

The final output of PigReuse is an optimized plan that contains (i) the operator nodes leading to *minimizing the cummulated cost* of all the queries in Q, while *producing, together, the same set of outputs* as the original Q, and (ii) equivalence nodes that represent *result sharing* of an operator node with other operators in Q. In the following sections, we describe each step of our reuse-based optimization algorithm in detail.

5.3.1 Equivalence-based merging

To join all detected equivalent expressions in Q, we build an AND-OR DAG, which we term *equivalence graph* (EG, in short); the construction is carried out in the spirit of previous optimization works [Gra93, RSSB00]. In the EG, an AND-node corresponds to an algebraic operation (e.g., selection, projection etc.). An OR-node o is introduced whenever a set of expressions $e_1, e_2, \ldots, e_k$ have been identified as equivalent; in the EG, o has as children the algebraic nodes at the roots of the expressions $e_1, e_2, \ldots, e_k$. In the following, we refer to AND-nodes as *operator nodes*, and OR-nodes as *equivalence nodes*.

Formally, we define an EG as follows.

Definition 15. *An equivalence graph (EG) is a DAG, defined by the pair* $(O \cup A \cup T_o, E)$*, such that:*

- $O \cup A \cup T_o$ *is the set of nodes:*
  - O *is the set of equivalence nodes.*
  - A *is the set of operator nodes.*
  - T_o *is the set of sink nodes.*
- $E \subseteq (O \times A) \cup (A \times O) \cup (A \times T_o)$ *is a set of directed edges such that:*
  - *Each node* $a \in A$ *has an in-degree of at least one, and an out-degree equal to one.*
  - *Each node* $o \in O$ *has an in-degree of at least one, and an out-degree of at least one.*
  - *Each node* $t_o \in T_o$ *has an in-degree of at least one.* ◇

Observe that in an EG, O nodes can only point to A nodes, while A nodes can only point to O nodes. In turn, T_o can only be pointed by A nodes.

An important point to stress here is that *equivalence nodes with more than one child amount to optimization opportunities* as they indicate that several operator nodes have a common (equivalent) child subexpression. In this case, we can choose the "best" way to compute the result of the subexpression among the choices given by the OR-node. The choice is based on a cost model, where the best plan corresponds to the plan with overall minimal cost. Optimal plan selection is discussed in detail in the next section.

Building the equivalence graph. To build the equivalence graph, we need to identify equivalent expressions within the input NRAB query set Q. We reuse the classical notion of query equivalence here, i.e., two expressions are *equivalent* iff their result is provably the same regardless of the data on which they are computed.

We build the EG in the following fashion. First, we create the EG *eg* with a single equivalence node o_s, i.e., the EG *source*. We take every NRAB query $q \in Q$ and perform a *breadth-first* traversal of its nodes. Each source node $s \in q$ is added to *eg*, and an edge (o_s, s) is created.

Subsequently, for each node n having the source node s as an input, we verify whether there exists a node n_{eg} in *eg*, such that the expression rooted in n is equivalent to the one rooted in n_{eg}.

- If such an equivalence is detected, we connect n to the equivalence node o that n_{eg} feeds.
- If no such equivalent node is found, n is added to *eg*, a new equivalence node o is added to *eg*, and an edge (n, o) is created. In either case, for each node n' that is a parent of n in the original query, n' is added to *eg* and an edge (o, n') is created. Within a set of equivalent nodes, each node is the root of a sub-DAG that represents a NRAB expression; the expressions corresponding to all these nodes are equivalent.

To check if two expressions $\mathscr{A}, \mathscr{A}'$ rooted at nodes n and n' are equivalent, we apply the known commutativity, associativity etc. laws that have been extensively studied for the bag relational algebra [BK90, GL95, PS96, RG03]. If one of the possible rewritings of $\mathscr{A}'$ (guaranteed to be equivalent to $\mathscr{A}'$ through the abovementioned prior work) matches the exact syntax of $\mathscr{A}$, then $\mathscr{A}$ and $\mathscr{A}'$ are equivalent, and they become children of the same equivalence node. Our equivalence search algorithm is sound but not complete; details about the equivalences we are capable of detecting can be found in Appendix B. Recall that the problem of checking equivalence of two arbitrary Relational Algebra expressions is undecidable [Sol79], and so the problem is for NRAB. Thus, no terminating equivalence checking algorithm exists for NRAB. However, as our experimental evaluation shows, the equivalences detected by PigReuse allow it to bring significant performance savings.

As mentioned above, the equivalent transformation rules we apply are those previously identified for NRAB, i.e., they only cover operators that have been previously defined as (extensions of) NRAB operators (i.e., $\uplus$, $-$, $\times$, ϵ, δ, *map*, σ, π, and $\bowtie$, see Table 5.1). As we will discuss in Section 5.4, we provide a set of new equivalent rewriting rules involving operators we introduced in this work (e.g., *cogroup* and outer join variants). These rules allow identifying more equivalences in an efficient way, and thus improve over the baseline PigReuse algorithm presented in this section.

Figure 5.6 depicts the EG corresponding to the NRAB DAGs q_1 to q_4 in Figure 5.5.b. In Figure 5.6, we use boxes to represent equivalence nodes, while sink nodes are represented by shadowed triangles. All the leaf nodes in the NRAB DAGs that correspond to the same *scan* operation (namely, nodes *A*, *B*, and *C*) feed the same equivalence node. The equi-joins coming from DAGs q_1 and q_2 on relations *A* and *B* over attributes *user* and *name* are also inputs to the same equivalence node.

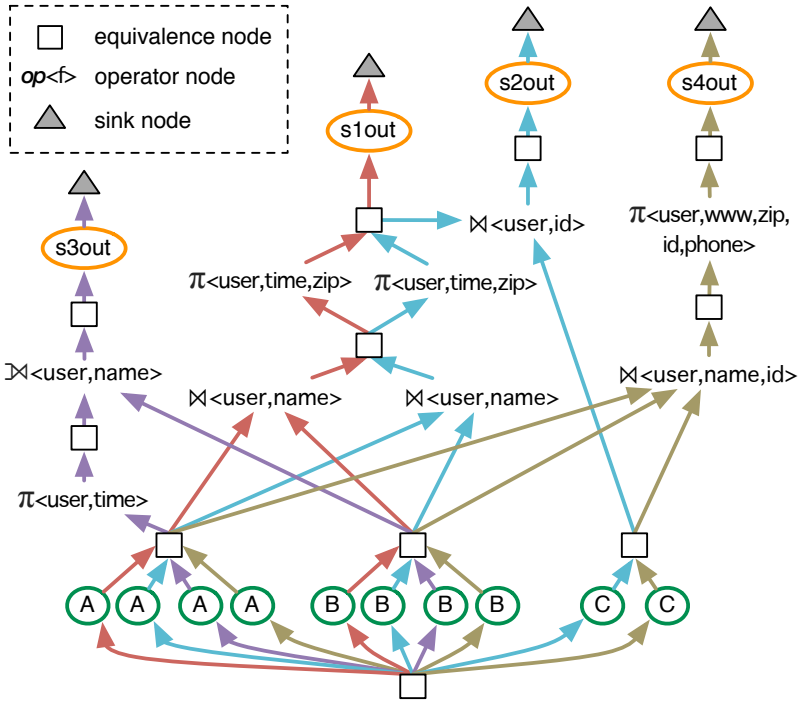


Figure 5.6: EG corresponding to NRAB DAGs q_1-q_4.

5.3.2 Cost-based plan selection

Once an EG has been generated from a set of NRAB queries, our goal is to find the best alternative plan (having the smallest possible cost) computing the same outputs as the original scripts, on any input instance.

We call the output plan a *result equivalence graph* (or REG, in short).

Definition 16. *A result equivalence graph (REG) with respect to an EG defined by* $(O \cup A \cup T_O, E)$ *is itself a DAG, defined by the pair* $(O^* \cup A^* \cup T_O, E^*)$ *such that:*

- $O^* \subseteq O$.
- $A^* \subseteq A$.
- *The set of sink nodes* T_o *is identical in EG and REG.*
- $E^* \subseteq E$.
- *Each sink node has an in-degree of exactly one.*
- *Each operator node in-degree in the REG is equal to its in-degree in the EG.*
- *Each equivalence node has an in-degree of exactly one, and an out-degree of at least one.* ◇

In the REG, we choose exactly one among the alternatives provided by each EG equivalence nodes; the REG produces the same outputs as the original EG, as all sink nodes are preserved.

Further, each REG can be straightforwardly translated into a NRAB DAG which is basically an executable Pig Latin expression. The latter expression is the one we turn to Pig for execution.

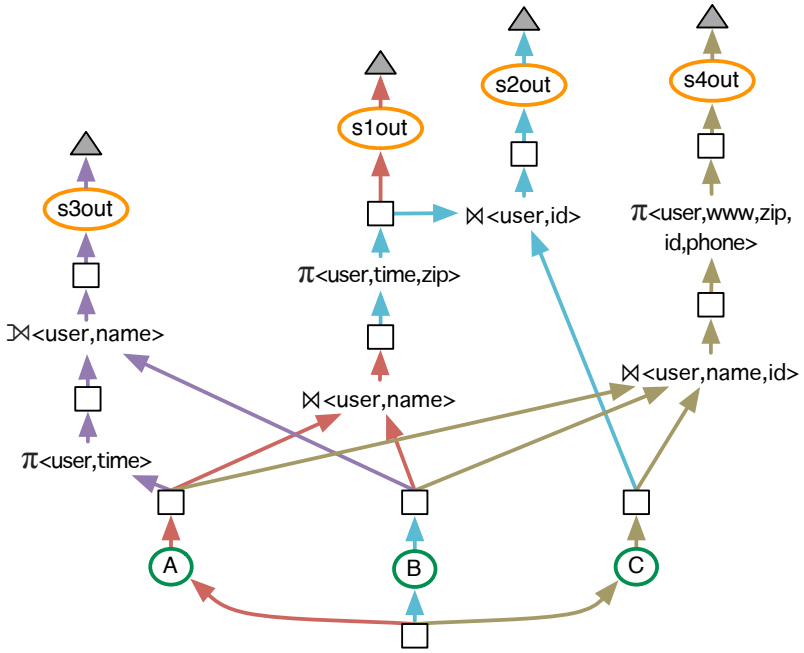


Figure 5.7: Possible REG for the EG in Figure 5.6.

The choice of which alternative to pick for each equivalence node is guided by a *cost function*, the overall goal being to minimize the global cost of the plan. We assign a cost (weight) to each edge $n_1 \rightarrow n_2$ in the EG, representing all the processing cost (or effort) required to fully build the result of n_2 out of the result of n_1.

Figure 5.7 shows a possible REG produced for the EG depicted in Figure 5.6. This REG could have been for instance obtained by using a cost function based on counting the operator nodes in the optimized script. In the REG, each equivalence node has exactly one input edge, i.e., the *scans* and other operator nodes are shared across queries, whenever possible. In Section 5.5, we consider different cost functions and compare them experimentally.

5.3.3 Cost minimization based on binary integer programming

We model the problem of finding the minimum-cost REG relying on *Binary Integer Programming* (BIP), a well-explored branch of mathematical optimizations that has been used previously to solve many optimization problems in the database literature [KKM13, YKL97]. Broadly speaking, a typical linear programming problem can be expressed as:

given a set of linear inequality constraints over a set of variables

find value assignments for the variables

such that the value of an objective function depending on these variables is minimized.

Such problems can be tackled by dedicated binary integer program *solvers*, some of which are by now extremely efficient, benefiting from many years of research and

$$\text{Minimize: } \mathscr{C} = \sum_{e \in E} \mathscr{C}_e x_e$$

subject to:

$$x_e \in \{0,1\} \qquad \forall e \in E \qquad (1)$$

$$\sum_{e \in E^{in}_{t_o}} x_e = 1 \qquad \forall t_o \in T_o \qquad (2)$$

$$\sum_{e \in E^{in}_{a}} x_e = x_{E^{out}_a} \times |E^{in}_a| \qquad \forall a \in A \qquad (3)$$

$$\sum_{e \in E^{in}_{o}} x_e = \max_{e \in E^{out}_o} x_e \qquad \forall o \in O \qquad (4)$$

Figure 5.8: BIP reduction of the optimization problem.

development efforts.

Generating the result equivalence graph. Given an input EG, for each of its nodes $n \in O \cup A \cup T_o$, we denote by E^{in}_n and E^{out}_n the sets of incoming and outgoing edges for n, respectively.

For each edge $e \in E$, we introduce a variable x_e, denoting whether or not e is part of the REG. Since in our specific problem formulation a variable x_e can only take values within $\{0,1\}$, our problem is formulated as a BIP problem.

Further, for each edge $e \in E$, we denote by $\mathscr{C}_e$ the cost $\mathscr{C}(e)$ assigned to e by some cost function $\mathscr{C}$. Importantly, the model we present in the following is independent of the chosen cost function.

Our optimization problem is stated in BIP terms in Figure 5.8. Equation (1) states that each x_e variable takes values in $\{0,1\}$. (2) ensures that every output is generated exactly once. (3) states that if the (only) outgoing edge of an operator node is selected, all of its inputs are selected as well. This is required in order for the algebraic operator to be capable of correctly computing its results. Finally, (4) states that if an equivalence node is generated, it should be used at least once, which is modeled by means of a *max* expression.

Since *max* is not directly supported in the BIP model, the actual BIP constraints which we use to express (4) are shown in Figure 5.9. These constraints encode the *max* constraint as follows. Equation (4.1) introduces a binary variable $d_{e \in E^{out}_o}$ used to model the *max* function. Equation (4.2) states that if an outgoing edge of an equivalence node is selected, then one of its incoming edges is selected too. (4.3) states that if no outgoing edge of an equivalence node is selected, then none of its incoming edges is selected. Further, (4.3) and (4.4) together ensure that if an outgoing edge of an equivalence node is selected, only one of its incoming edges will be selected. Observe that we can model the *max* function in this fashion since in equation (4), *max* is computed over a set of inputs whose values are in $[0,1]$.

$$d_{e \in E_o^{out}} \in \{0,1\} \qquad \forall o \in O \quad (4.1)$$

$$\sum_{e \in E_o^{in}} x_e \geq x_{e \in E_o^{out}} \qquad \forall o \in O \quad (4.2)$$

$$\sum_{e \in E_o^{in}} x_e \leq (x_e - d_e + 1)_{e \in E_o^{out}} \qquad \forall o \in O \quad (4.3)$$

$$\sum_{e \in E_o^{out}} d_e = 1 \qquad \forall o \in O \quad (4.4)$$

Figure 5.9: BIP representation of the *max* constraint.

ϵ	*map*	σ	π	*mapconcat*	$\uplus$	$\times$	*cogroup*	$\bowtie$	$\rtimes$	$\ltimes$	$\overline{\rtimes}$
■	□	□	□	□	■	□◊	□◊	□◊	□◊	□◊	□◊

□ *Child* π *operator can be swapped with the parent operator, iff none of the fields used by the parent operator is projected by* π.
◊ *Child* π *operator can be swapped with the parent operator only after rewriting the original* π *operator.*
■ *Child* π *operator cannot be swapped with the parent operator.*

Figure 5.10: Reordering and rewriting rules for π.

5.4 Effective reuse-based optimization

In this section, we introduce a set of techniques for identifying and exploiting additional subexpression factorization opportunities that go beyond those that are possible with the standard NRAB operators. The three extensions we bring to the basic PigReuse algorithm are: normalization (Section 5.4.1), join decomposition (Section 5.4.2), and aggressive merge (Section 5.4.3).

5.4.1 Normalization

Normalization of the input NRAB DAGs is carried out by *reordering* π operator nodes as follows: we push them away from *scan* operators or closer to *store* operators. We do this by visiting all operator nodes in a NRAB DAG, starting from a *scan*, and by moving each π operator up one level at a time. As we will shortly illustrate, *pushing projections up increases the chances to find equivalent subexpressions*.

Figure 5.10 spells out the conditions under which a π can be swapped with its parent operator. Each column in the topmost row represents a parent operator with which the child π may be swapped, and the value of each cell represents different conditions under which the swap is possible. For example, a child π can be swapped with a parent σ, iff the selection predicate does not carry over the attributes projected in π.

A special case is the *cogroup* operator. Since *cogroup* nests the input relations,

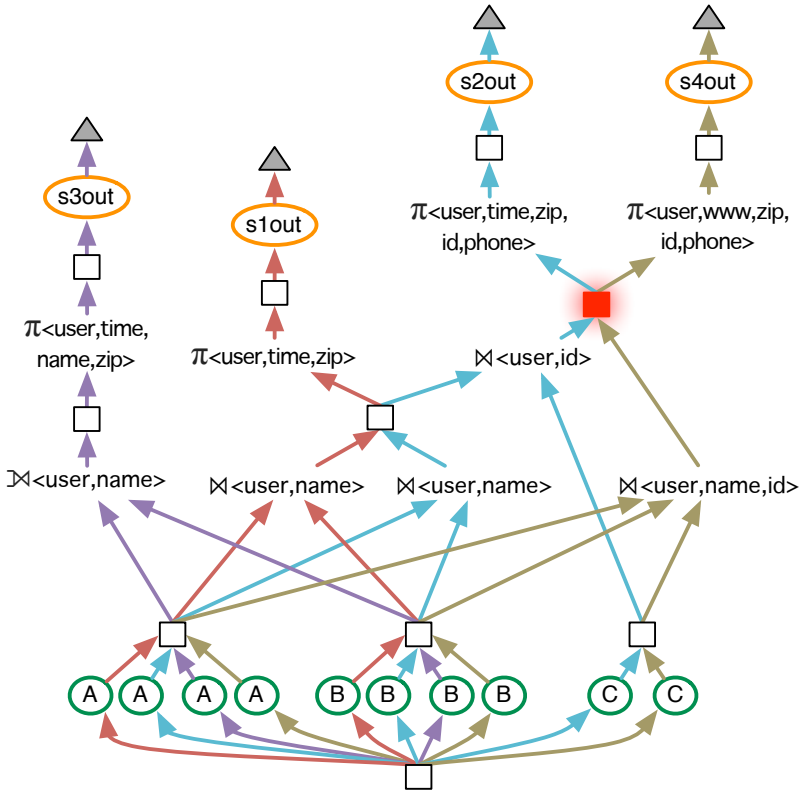


Figure 5.11: EG generated by PigReuse on the *normalized* NRAB DAGs q_1-q_4.

reordering π with this operator requires complex rewriting. In particular, we will rewrite it into a *map* that applies the projection π on the bag of tuples corresponding to the input relation. *map* operators containing only combinations of *map* and π can still be pushed up following the conditions in Figure 5.10. This means that, in general, during normalization, one may need to introduce *map* operators nested more than two levels deep. Although the Pig Latin query language does not allow more than two levels of nested `FOREACH` expressions, our NRAB representation *map* allows it; furthermore, as we have found examining the code for executable plans within the Pig Latin engine, more than two levels of nesting *are* supported at the level of the execution engine [2].

Observe that operators such as ϵ or $\uplus$ restrict the possibilities of moving π operators across the DAG. It turns out also that they do not commute with the other algebraic operators; we term these "unmovable" operators, *bordering* operators in the sense that they raise borders to the moving of π across the DAG.

After our reuse-based algorithm produces the optimized REG, to avoid the performance loss incurred by manipulating many attributes at all levels (due to the pulling up of the projections), we push the π operators back, as close to the *scan* as possible.

2. The class `pig.newplan.logical.relational.LOForEach`, representing the `FOREACH` operator, has a field called `innerPlan` which in our tests could contain another `LOForEach` and so on on several levels. The purpose of introducing the language-level restriction may have been to prevent programmers from writing deeply-nested loops whose performance could be poor.

$$\frac{\begin{array}{c}\mathscr{A}_1 := cogroup\langle a_1, a_2, \ldots, a_n\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2, \ldots, \underline{\mathsf{var}}_n) \\ \mathscr{A}_2 := map\langle \delta(\underline{\mathsf{var}}_1) \times \delta(\underline{\mathsf{var}}_2) \times \ldots \times \delta(\underline{\mathsf{var}}_n)\rangle(\mathscr{A}_1)\end{array}}{\bowtie \langle a_1, a_2, \ldots, a_n\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2, \ldots, \underline{\mathsf{var}}_n) = \mathscr{A}_2} \quad \text{(IJ)}$$

$$\frac{\begin{array}{c}\mathscr{A}_1 := cogroup\langle a_1, a_2\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2) \\ \mathscr{A}_2 := map\langle \delta(\underline{\mathsf{var}}_1) \times \delta(empty(\underline{\mathsf{var}}_2)?\{\{\bot\}\} : \underline{\mathsf{var}}_2)\rangle(\mathscr{A}_1)\end{array}}{⟕ \langle a_1, a_2\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2) = \mathscr{A}_2} \quad \text{(LOJ)}$$

$$\frac{\begin{array}{c}\mathscr{A}_1 := cogroup\langle a_1, a_2\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2) \\ \mathscr{A}_2 := map\langle \delta(empty(\underline{\mathsf{var}}_1)?\{\{\bot\}\} : \underline{\mathsf{var}}_1) \times \delta(\underline{\mathsf{var}}_2)\rangle(\mathscr{A}_1)\end{array}}{⟖ \langle a_1, a_2\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2) = \mathscr{A}_2} \quad \text{(ROJ)}$$

$$\frac{\begin{array}{c}\mathscr{A}_1 := cogroup\langle a_1, a_2\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2) \\ \mathscr{A}_2 := map\langle \delta(empty(\underline{\mathsf{var}}_1)?\{\{\bot\}\} : \underline{\mathsf{var}}_1) \times \\ \qquad \delta(empty(\underline{\mathsf{var}}_2)?\{\{\bot\}\} : \underline{\mathsf{var}}_2)\rangle(\mathscr{A}_1)\end{array}}{⟗ \langle a_1, a_2\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2) = \mathscr{A}_2} \quad \text{(FOJ)}$$

Figure 5.12: Decomposing JOIN operators.

As our normalization algorithm may rewrite π operators using *map*, we extended the Pig Latin optimizer to support the (unnesting) rewriting of such cases, so that the π can be pushed back down through the plan. Recall that even if they cannot be pushed back down, the resulting plan (no matter how many levels the π operators are nested) will be executable by the Pig engine.

To illustrate the advantages of our normalization phase, Figure 5.11 shows the EG generated by PigReuse over the *normalized* NRAB DAGs q_1 to q_4. Comparing this EG with the one shown in Figure 5.6, we see that due to the swapping of the π operator corresponding to q_2, our algorithm can identify an additional common subexpression between q_2 and q_4, by determining the equivalence between the joins over A, B, and C; the corresponding equivalence node is highlighted in Figure 5.11.

5.4.2 Join decomposition

The semantics of Pig Latin 's join operators e.g., $\bowtie$, ⟕, ⟖, or ⟗ allow rewriting (or *decomposing*) these operators into combinations of *cogroup* and *map* operators. The advantage of decomposing the joins in this way is that the result of the *cogroup* operation, which does the heavy-lifting of assembling groups of tuples from which the *map* will then build join results, can be shared across different kinds of joins. The *map* will be different in each case depending on the join type, but the most expensive component of computing the join, namely the *cogroup*, will be factorized. Further, there is no noticeable performance difference between executing a certain join or its decomposed rewritten version.

Figure 5.12 shows the decomposition rules that are applied on the input NRAB DAGs.

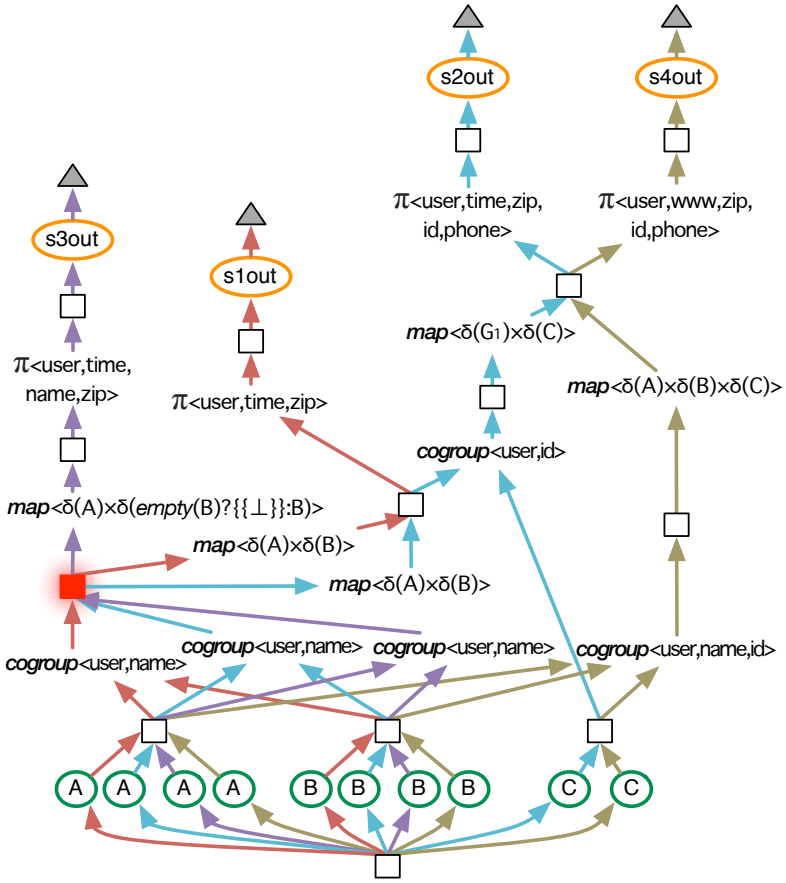


Figure 5.13: EG generated by PigReuse on the *normalized* and *decomposed* NRAB DAGs q_1-q_4.

Rule (IJ) rewrites an inner equi-join ⋈ into two operators. The first one is a *cogroup* on the attributes used by the join predicate. The second one is a *map* that does the following for each input tuple: (i) project each bag of tuples corresponding to the *cogroup* input relations; (ii) apply a δ operation on each of those bags; and (iii) perform a cartesian product among the tuples resulting from unnesting those bags. Observe that if a bag is empty, e.g., the input relation did not contain any value for the given grouping value, the δ operator does not produce any tuple, and thus the tuples from the other bags for the given tuple are discarded. Thus, this rewriting produces the exact same result as the original ⋈ operator.

The rest of the rules use the aggregation function *empty*, that checks if an input bag is empty. For instance, the expression $empty(\underline{\text{var}})?\{\{\bot\}\} : \underline{\text{var}}$ is a conditional assignment, that is: if var is empty, a bag with a null tuple ($\bot$), i.e., a tuple whose values are bound to null values, conforming to var schema is assigned, otherwise the bag var is assigned.

Rule (LOJ) rewrites a left outer join ⟕ into a *cogroup* on the attributes used by the join predicate, followed by a *map* operator that (i) unnests the bag associated to the left input of the *cogroup*; (ii) if the bag associated to the right input ($\underline{\text{var}}_2$)

$$\frac{\begin{array}{c}\mathscr{A}_1 := cogroup\langle a'_1, a'_2, \ldots, a'_n\rangle(\underline{\mathsf{var}}'_1, \underline{\mathsf{var}}'_2, \ldots, \underline{\mathsf{var}}'_n)\\ \mathscr{A}_2 := \pi\langle group, \underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_k\rangle(\mathscr{A}_1)\\ \mathscr{A}_3 := \sigma\langle\neg(empty(\underline{\mathsf{var}}_1) \wedge \ldots \wedge empty(\underline{\mathsf{var}}_k))\rangle(\mathscr{A}_2)\end{array}}{\begin{array}{c}cogroup\langle a_1, \ldots, a_k\rangle(\underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_k) = \mathscr{A}_3 \mid\\ \underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_k \subset \underline{\mathsf{var}}'_1, \underline{\mathsf{var}}'_2, \ldots, \underline{\mathsf{var}}'_n \wedge a_1, \ldots, a_k \subset a'_1, a'_2, \ldots, a'_n\end{array}} \quad \text{(CG-CG)}$$

$$\frac{\begin{array}{c}\mathscr{A}_1 := cogroup\langle a'_1, a'_2, a'_3, \ldots, a'_n\rangle(\underline{\mathsf{var}}'_1, \underline{\mathsf{var}}'_2, \underline{\mathsf{var}}'_3, \ldots, \underline{\mathsf{var}}'_n)\\ \mathscr{A}_2 := map\langle\delta(\underline{\mathsf{var}}_1) \times \delta(\underline{\mathsf{var}}_2) \ldots \times \delta(\underline{\mathsf{var}}_k)\rangle(\mathscr{A}_1)\end{array}}{\begin{array}{c}\bowtie\langle a_1, a_2, \ldots, a_k\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2, \ldots, \underline{\mathsf{var}}_k) = \mathscr{A}_2 \mid\\ \underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2, \ldots, \underline{\mathsf{var}}_k \subset \underline{\mathsf{var}}'_1, \underline{\mathsf{var}}'_2, \underline{\mathsf{var}}'_3, \ldots, \underline{\mathsf{var}}'_n \wedge a_1, a_2, \ldots, a_k \subset a'_1, a'_2, a'_3, \ldots, a'_n\end{array}} \quad \text{(IJ-CG)}$$

$$\frac{\begin{array}{c}\mathscr{A}_1 := cogroup\langle a'_1, a'_2, a'_3, \ldots, a'_n\rangle(\underline{\mathsf{var}}'_1, \underline{\mathsf{var}}'_2, \underline{\mathsf{var}}'_3, \ldots, \underline{\mathsf{var}}'_n)\\ \mathscr{A}_2 := map\langle\delta(\underline{\mathsf{var}}_1) \times \delta(empty(\underline{\mathsf{var}}_2)?\{\{\bot\}\} : \underline{\mathsf{var}}_2)\rangle(\mathscr{A}_1)\end{array}}{\begin{array}{c}\rtimes\langle a_1, a_2\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2) = \mathscr{A}_2 \mid\\ \underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2 \subset \underline{\mathsf{var}}'_1, \underline{\mathsf{var}}'_2, \underline{\mathsf{var}}'_3, \ldots, \underline{\mathsf{var}}'_n \wedge a_1, a_2 \subset a'_1, a'_2, a'_3, \ldots, a'_n\end{array}} \quad \text{(LOJ-CG)}$$

$$\frac{\begin{array}{c}\mathscr{A}_1 := cogroup\langle a'_1, a'_2, a'_3, \ldots, a'_n\rangle(\underline{\mathsf{var}}'_1, \underline{\mathsf{var}}'_2, \underline{\mathsf{var}}'_3, \ldots, \underline{\mathsf{var}}'_n)\\ \mathscr{A}_2 := map\langle\delta(empty(\underline{\mathsf{var}}_1)?\{\{\bot\}\} : \underline{\mathsf{var}}_1) \times \delta(\underline{\mathsf{var}}_2)\rangle(\mathscr{A}_1)\end{array}}{\begin{array}{c}\ltimes\langle a_1, a_2\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2) = \mathscr{A}_2 \mid\\ \underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2 \subset \underline{\mathsf{var}}'_1, \underline{\mathsf{var}}'_2, \underline{\mathsf{var}}'_3, \ldots, \underline{\mathsf{var}}'_n \wedge a_1, a_2 \subset a'_1, a'_2, a'_3, \ldots, a'_n\end{array}} \quad \text{(ROJ-CG)}$$

Figure 5.14: Rules for aggressive merge.

is empty, it replaces it with a bag with a null tuple, otherwise it keeps the bag as it is; (iii) unnests the bag resulting from the previous operation; and (iv) performs a cartesian product on the tuples resulting from the δ operations in order to generate the $\rtimes$ result. Rule (ROJ) rewrites a right outer join $\ltimes$ in a similar fashion.

Finally, rule (FOJ) rewrites a full outer join $\bowtie$ following the same principle as for the two previous operators. The difference is that in (FOJ) we check the bags from both inputs by means of the *empty* function.

Figure 5.13 shows the EG generated by PigReuse after applying *normalization and decomposition* to the NRAB DAGs q_1 to q_4. One can observe that the decomposition of the $\bowtie$ operators from q_1 and q_2, and the $\rtimes$ operator from q_3 leads to an additional sharing opportunity, as the result of the *cogroup* on attributes *user* and *name* can be shared by the subsequent *map* operations (highlighted equivalence node).

5.4.3 Aggressive merge

The last extension we propose is based on the observation that it is possible to derive the results of a $\bowtie$ or *cogroup* operator from the results of a *cogroup*$'$ operator, as long as the former relies on *a subset* of the input relations and attributes of *cogroup*$'$. This means that these rewritings rely on the notion of *cogroup containment*. In particular, this entails checking the containment relationship between respective sets of

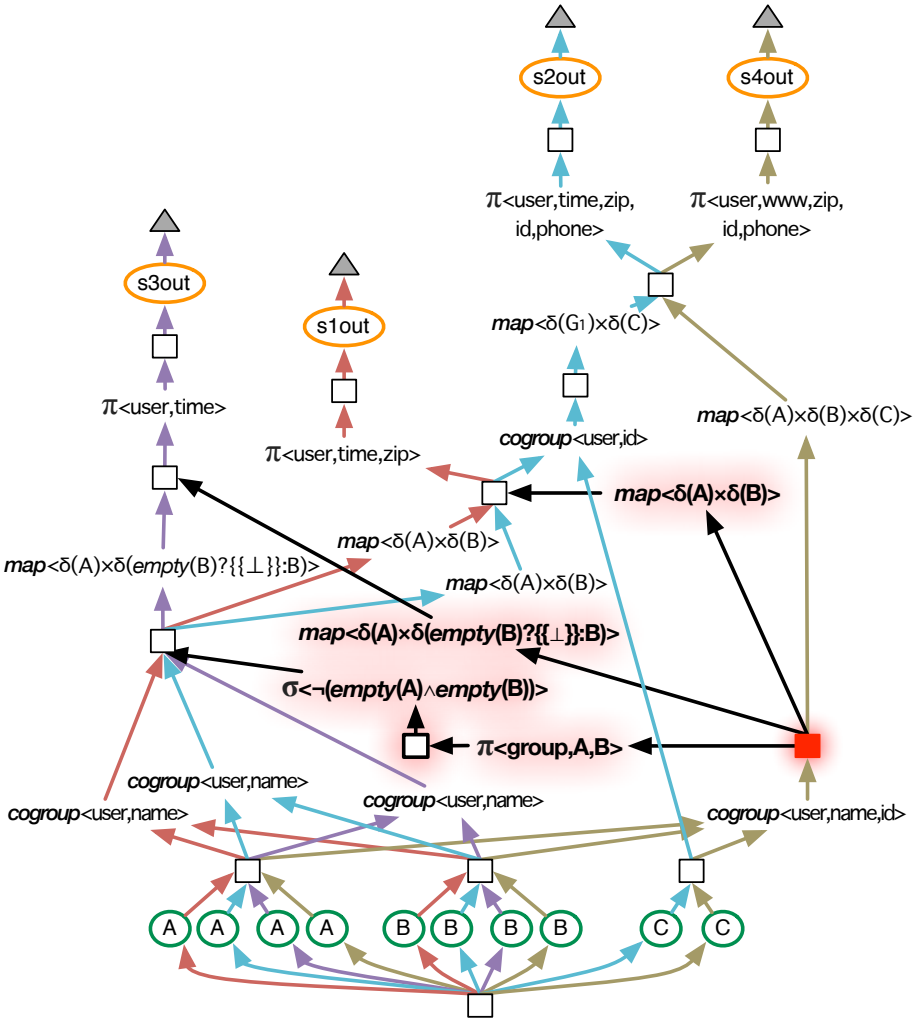


Figure 5.15: EG generated by PigReuse applying *aggressive merge* on the *normalized* and *decomposed* NRAB DAGs q_1-q_4.

input relations and attributes. Then, in order to generate the result of the original $\bowtie$ or *cogroup* operator, we add the appropriate operator on top of *cogroup′*; this can be seen as a limited instance of query rewriting using views, where *cogroup′* plays the role of a view. In contrast to the previous extensions that are applied on the input NRAB DAGs, *aggressive merge* is applied while creating the EG.

Figure 5.14 shows the rewritings considered by our aggressive merge algorithm.

Rule (CG-CG) states that if a query contains a *cogroup′* operator with two or more input relations, any other *cogroup* (with at least one input relation, part of the *cogroup′* input) can be derived from the previous one in the following fashion. First, a π operator projects the subset of attributes that are needed for the result of the *cogroup* operator. Then, a σ operator discards the tuples where *all* the bags associated to each input relation are empty.

Rules (IJ-CG), (LOJ-CG), and (ROJ-CG) are similar to those shown in Figure 5.12; the only difference is that the *map* operators take only a subset of the bag attributes

in the original *cogroup*. Note that we do not have a rule for the ⟕ operator since we are able to generate its output directly from the result of the *cogroup*.

Figure 5.15 depicts the EG produced by PigReuse using the aggressive merge extensions, when normalization and decomposition has been applied to the NRAB plans q_1-q_4. The new connections created by aggressive merge are highlighted. The figure shows how the results for the *cogroup*, ⋈, and ⟕ operators on A and B relations are derived from the *cogroup* operator on A, B, and C.

5.5 Implementation and experimental evaluation

We have implemented PigReuse, our reuse-based optimization approach, in Java 1.6. The source code amounts to about 8000 lines and 50 classes. It works on top of Apache Pig 0.12.1 [Piga], which relied on the Hadoop platform 1.1.2 [Had]. The cost-based plan selection algorithm (Section 5.3.2) uses the Gurobi BIP solver 5.6.2 [Gur].

Section 5.5.1 describes our experimental setup. Then, Section 5.5.2 presents the two alternative cost functions that we have implemented and experimented with; recall that while the cost function does impact the configuration chosen by the BIP solver, our approach and algorithms are independent of the cost function chosen. Finally, Section 5.5.3 presents our experimental results.

5.5.1 Experimental setup

Deployment. All our experiments run in a cluster of 8 nodes connected by a 1GB Ethernet. Each node has a 2.93GHz Quad Core Xeon processor and 16GB RAM. The nodes run Linux CentOS 6.4. Each node has two 600GB SATA hard disks where HDFS is mounted.

Setup. For validation, we used data sets and scripts provided by the PigMix [Pigb] PigLatin performance benchmark. In particular, we created a `page_views` input file of 250 million rows; the benchmark includes other input files, which are based on the `page_views` file, and are much smaller than this one. The total size of the data set amounted to approximately **400 GB** before the 3-way replication applied by HDFS.

We run our algorithm with two different workloads. The first one (denoted W_1) comprises 12 scripts taken directly from the PigMix benchmark, namely l_2-l_7 and l_{11}-l_{16}); these only use operators supported by our current implementation, e.g., `JOIN`, `COGROUP`, `FILTER` etc. Each script has on average 7 operators. The second workload (W_2) includes W_1, to which we add 8 extra scripts which feature many `JOIN` flavours, `COGROUP` on many relations etc. These scripts are created to give opportunities to validate our algorithm on a wider variety of operators. Further details about these workloads can be found in Appendix C.

5.5.2 Cost functions and experiment metrics

We now present the two cost functions that are implemented currently in PigReuse, focusing on the number of logical operators, and the number of MapReduce jobs, respectively. Although more elaborated cost functions can be envisioned, these two already lead to considerable gains due to reuse, as our experiments shortly show.

Operator-based cost function. A first cost function characterizing the effort required by the evaluation of a batch of Pig Latin scripts is the number of operators in the equivalent NRAB expression eventually evaluated, that is:

$$\mathscr{C}_e = 1 \quad \forall e \in E_a^{out},\ \forall a \in A$$
$$\mathscr{C}_e = 0 \quad \text{for all the rest}$$

Above, we assign a cost of 1 to the execution of every algebraic operator a, and we attach this cost to its outgoing edge. All the other edges, i.e., incoming edges to an operator node, have a cost of 0.

MapReduce jobs-based cost function. Our second cost function is closely related to the Pig execution engine on top of MapReduce. The function minimizes the MapReduce jobs needed to compute the results of the input Pig Latin scripts, as some groups of operators are executed by Pig as part of the same job. For instance, σ, π, and *map* do not generate a new MapReduce job, which is very convenient for our *decomposition* and *aggressive merge* extension techniques that introduce these operators quite aggressively when rewriting.

While these cost functions are used by our reuse algorithm, we also use the following two standard metrics to quantify the performance of executing a PigLatin script workload:

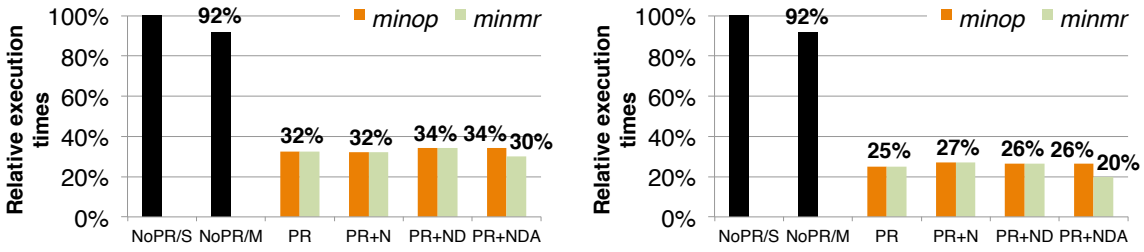
Execution time is the wallclock time measured from the moment when the scripts are submitted to the Pig engine, until the moment their execution is completely finished;

Total work is the sum of the effort made on all the nodes, i.e., the total CPU time as returned by logs of the MapReduce execution engine.

5.5.3 Experimental results

We now study the benefits brought by the optimizations proposed in this work. The reported results are averaged over three runs.

Figure 5.16 shows the effectiveness of our baseline PigReuse algorithm (PR), PigReuse with *normalization* (PR+N), PigReuse with *normalization* and *decomposition* (PR+ND), and PigReuse applying all our extensions including *aggressive merge* (PR+NDA). The figure shows relative values for the execution time and total work metrics. The cost function that minimizes the total number of operators in the EG is denoted by *minop*, while the cost function that minimizes the total number of MapReduce jobs is denoted by *minmr*.



(a) Execution time

(b) Total work

Figure 5.16: PigReuse evaluation using workload W_1 (left) and W_2 (right).

In Figure 5.16.a, we notice that our PigReuse algorithms reduce the total execution time by more than 70% on average. Two alternative executions without PigReuse are shown. In the first one (NoPR/S), we execute *sequentially* every script in each workload using a single Pig client. In the second one (NoPR/M), we use multiple Pig clients that send *concurrently* the jobs resulting from the scripts to MapReduce. As it can be seen, the execution time for the second variant is lower as jobs resulting from multiple scripts are scheduled together, and thus the cluster usage is maximized. However, observe that the total work (Figure 5.16.b) increases for the multi-client alternative. In a nutshell, this is because the number of slots needed for map tasks is very large, so the scheduler cannot create a lot of overlapping among the map phases of multiple queries. So their execution remains quite sequential.

For the workloads we considered, our extensions reduced the total work over the baseline PigReuse algorithm (Figure 5.16.b). However, the same did not happen for the execution time in all cases (Figure 5.16.a). The reason is that some of the resulting plans entailed more effort but consisted of less execution steps that could be parallelized easier by the MapReduce engine.

When *aggressive merge* was applied, the execution time and the total work decreased only if the *minmr* cost function was used. The reason is that if the *minop* function is used, PigReuse generates the same REG for PR+ND and PR+NDA, namely, the REG with the minimum number of operators. However, if the *minmr* cost function is used, PigReuse chooses an alternative plan that executes faster even though it has more operators.

Table 5.2 provides some important metrics concerning the EGs and REGs created by PigReuse algorithm. As shown in the table, the algorithm reduces the total number of logical operators by an average of 30%, using any of the two cost functions. In particular, the REG generated by PigReuse using the *minop* or *minmr* cost functions has

		PigReuse			
		PR	**PR+N**	**PR+ND**	**PR+NDA**
W_1	EG - Equivalent nodes (#)	58	59	60	62
	EG - Operator nodes (#)	83	79	83	87
	REG (*minop*) - Operator nodes (#)	57	58	59	59
	REG (*minmr*) - Operator nodes (#)	57	58	59	60
W_2	EG - Equivalent nodes (#)	74	82	83	88
	EG - Operator nodes (#)	135	125	131	143
	REG (*minop*) - Operator nodes (#)	73	81	82	82
	REG (*minmr*) - Operator nodes (#)	73	81	82	85

Table 5.2: Reuse-based optimization details for workloads W_1 and W_2.

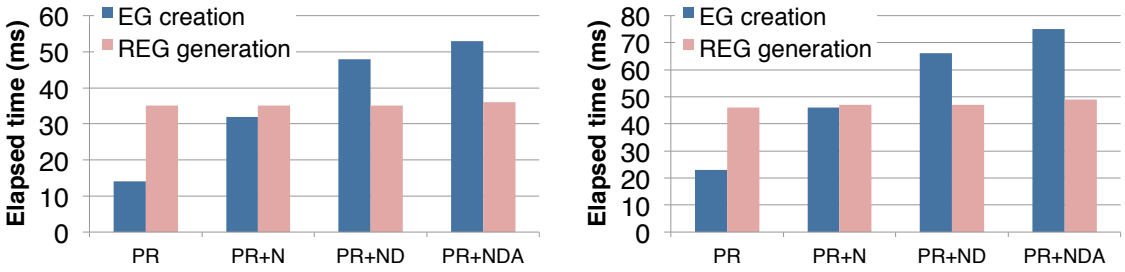


Figure 5.17: PigReuse compile time overhead for workloads W_1 (left) and W_2 (right).

the same number of operators, except when *aggressive merge* is used (PR+NDA). The reason is that all the connections that we establish through the aggressive merge strategy do not result in extra MapReduce jobs. Thus, using that strategy and the *minmr* cost function, a plan that contains more nodes but translates into less MapReduce jobs is selected. As we have seen before, this alternative plan leads to considerable execution time savings.

Finally, Figure 5.17 shows the total compile time overhead of using PigReuse. EG creation time includes the time to generate the EG, i.e., identifying equivalent expressions and merging them, and the time to apply our extensions to the algorithm (if any). We can observe that the EG creation time increases as the extensions to the baseline PigReuse are applied. On the other hand, the time to generate the REG is almost constant among all the strategies. It is important to note that the total optimization time stays below 125ms in all cases, which is less than 0.007% for all these workloads whose running time ranges from 28 minutes to 2 hours 35 minutes. This demonstrates that PigReuse obtains a remarkable execution time improvement with a negligible overhead, demonstrating its practical interest.

5.6 Related works

Our work relates to several areas of existing research.

Relational multi-query optimization. Early works on multi-query optimization

(MQO) [Jar85, Sel88] sought to improve the performance of query batches featuring common subexpressions, thus they are the most directly related to our work. These works proposed exhaustive, expensive optimization algorithms which were not integrated with existing system optimizers. [RSSB00] was the first to integrate MQO into a Volcano-style optimizer, while [ZLFL07] presents a completely integrated MQO solution also comprising the maintenance and exploitation of materialized views. Finally, the recent [SLZ12] presents a MQO approach taking into account the physical requirements (e.g., data partitioning) of the consumers of common sub-expressions in order to propose globally optimal execution plans.

To the best of our knowledge, equivalence or containment-based optimizations on the NRAB representation of Pig Latin scripts has not been studied before. We argue that our formalization into NRAB (which we are the first to provide) lays the adequate foundation for our reuse-based optimization, with correctness guarantees.

Reuse-based optimizations on MapReduce. Multiple works have focused on avoiding redundant processing for a batch of MapReduce jobs by sharing their (intermediate) results [AKO08, EA12, LSH$^+$14, NPM$^+$10, WC13]. In contrast to the PigReuse approach, these works either (i) need some information about the MapReduce job semantics in order to be efficient [EA12, LSH$^+$14], or (ii) their detected reuse-based optimization opportunities are limited to inputs and outputs of the mappers and reducers [AKO08, NPM$^+$10, WC13]. Our PigReuse algorithm works on the semantic representation of Pig Latin scripts. This enables complex reuse-based optimizations, e.g., those based on rewritings of expressions, and connects the NRAB representation to the real execution effort through a customizable cost function.

Single query optimization for MapReduce jobs. Recent works have proposed optimizations for MapReduce jobs [HB11, JCR11]. Our approach is orthogonal and complimentary to these optimizations, as we can detect common subexpressions among batches of Pig Latin queries at the higher level, and then these optimizations may be applied on the MapReduce jobs generated by the Pig engine.

Optimization using integer programming. Integer programming has been used before to model different optimization problems in data management systems, e.g., in materialized view selection and maintenance [YKL97], or optimal utilization of materialized views in publish/subscribe systems [KKM13]. Although our optimization goal is different, we got inspiration from these works to model the cost-based plan selection using integer programming.

5.7 Summary

This chapter has presented a novel approach for identifying and reusing repeated subexpressions occurring in Pig Latin scripts. In particular, we lay the foundation of our reuse-based algorithms by formalizing the semantics of the Pig Latin query language with Extended Nested Relational Algebra for Bags, that is extended accordingly to accommodate the semantics of Pig Latin operators. Our PigReuse algorithm

identifies sub-expression merging opportunities, and selects the best ones to merge based on a cost-based search process implemented with the help of a linear program solver. The output of our algorithm is a merged script reducing a given cost function, e.g., the number of operators or the number of MapReduce jobs required for execution. Our experimental results demonstrate the value of our reuse-based algorithms and optimization strategies.

Chapter 6

Conclusion and Future Work

As the volume and heterogeneity of Web data continue increasing very rapidly, the need for efficient systems to manage and extract information from it becomes more evident.

In this thesis, we have explored the performance and cost of warehousing Web data into commercial cloud infrastructures, which are so easily accessible nowadays. Further, we have explored the parallelization and optimization of query languages for processing Web data over these scalable infrastructures, in order to tackle the data management challenges that we face today.

6.1 Thesis summary

In this thesis, we have focused on three different problems that we summarize below.

Warehousing Web data using commercial cloud services. We presented AMADA, an architecture for warehousing Web data using commercial cloud services.

- We proposed a generic architecture for large-scale warehousing of complex Web data using commercial cloud services.
- We modeled the monetary costs associated to the exploitation of the warehouse.
- We investigated the usage of content indexing for tree-shaped data, and showed how indexes served not only as a tool to improve query performance, but also as a mean to reduce the warehouse associated monetary costs.
- We presented a concrete implementation of our architecture on top of the Amazon Web Services, and presented experimental results evaluating the system performance as well as its cost.

Parallelizing XQuery execution. We presented PAXQuery, a massively parallel processor of XML queries.

- We presented a methodology for massively parallel evaluation of XQuery without any effort from the user.
- We provided translation algorithms from the algebraic representation of a large fragment of XQuery to a parallel programming model, namely the the *PAral-*

lelization ConTracts (or PACT).
- We modeled the translation of complex flavors of join operators that were needed to execute efficiently XQuery using PACT. The interest of this translation goes beyond the context of XQuery evaluation, as it can be adopted to compile programs expressed in other high-level languages into PACT.
- We fully implemented our translation technique into our PAXQuery platform. We evaluated the scalability and performance of PAXQuery, and demonstrated that our approach outperforms other competitor systems.

Reuse-based optimization for Pig Latin. We considered the problem of identifying and reusing common sub-expressions occurring in Pig Latin scripts.
- We formalized the representation of Pig Latin scripts based on an existing well-established algebraic formalism, specifically Nested Relational Algebra for Bags (NRAB), providing a formal foundation for identifying correctly common expressions in batches of Pig Latin scripts.
- We proposed PigReuse, a multi-query optimization algorithm that merges equivalent sub-expressions it identifies in directed acyclic graphs of NRAB operators corresponding to PigLatin scripts. After identifying such reutilization opportunities, PigReuse produces a merged plan where redundant computations have been eliminated.
- We presented extensions to our baseline PigReuse optimization algorithm to increase the number of common subexpressions it detects.
- We presented an experimental evaluation of PigReuse and its extensions, showing the efficiency of our approach, which reduced the execution time of batches of PigLatin scripts in more than 80% for some cases.

6.2 Perspectives

The efficient large-scale processing of Web data in massively distributed environments is a very active area of research. We outline below various avenues for future work.

Warehousing Web data using commercial cloud services. An interesting direction for this work is to explore query optimization based on multiple objectives [TK14]. These objectives could be conflicting, such as minimizing execution time as well as monetary costs. A possible outcome of this effort could be an *advisor*, which based on specified user requirements, comes up with the best compromise that guarantees that those requirements are met. Further, AMADA's implementation could be extended to support Google and Azure commercial cloud platforms. Then, the previous decision would not consist only on deciding *which indices to create* or *how to execute a given query*, but on deciding *which cloud services provider* we should use in order to do it.

Parallelizing XQuery execution. An interesting avenue of work is to introduce new second order functions in the PACT model, and extend Stratosphere optimizer to efficiently support them. Some examples may include contracts to perform range

partitioning or streaming (window) processing. Then, we could enrich the XQuery fragment supported in this work.

Further, the support for efficient nesting processing in these massively parallel execution frameworks remains a challenging problem. It would be interesting to explore more complex models that would allow automatic and seamlessly parallelization of this nested computations.

Reuse-based optimization for Pig Latin. Our current focus in this project is on proposing richer cost functions, so PigReuse can adapt the plan selection to the query workload more precisely.

Since the emergence of MapReduce, there is an increasing corpora of work that focuses on optimizing data processing in massively parallel frameworks [LOOW14, DN14]. An interesting direction is to extend our PigReuse algorithm to be able to select and maintain intermediary results that could be used in future scripts executions. Observe that the side-effect of this extension is that we could provide *smart selective* fault-tolerance in MapReduce, in contrast to the costly materialization of all intermediary results that takes place in its current implementation. Further, we could explore the creation of internal structures to access more efficiently these results at runtime.

Finally, an additional interesting direction would be to explore extensions for our PigReuse algorithm to efficiently support multi-user environments.

Bibliography

[AABCR+12] Andrés Aranda-Andújar, Francesca Bugiotti, Jesús Camacho-Rodríguez, Dario Colazzo, François Goasdoué, Zoi Kaoudi, and Ioana Manolescu. AMADA: Web Data Repositories in the Amazon Cloud (demo). In *CIKM*, 2012.

[Aba09] Daniel J. Abadi. Data Management in the Cloud: Limitations and Opportunities. *IEEE Data Eng. Bull.*, 2009.

[ABC+03] Serge Abiteboul, Angela Bonifati, Grégory Cobéna, Ioana Manolescu, and Tova Milo. Dynamic XML Documents with Distribution and Replication. In *SIGMOD*, 2003.

[ABE+14] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, MatthiasJ. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The Stratosphere platform for big data analytics. *VLDB Journal*, 2014.

[ABM05] Andrei Arion, Véronique Benzaken, and Ioana Manolescu. XML Access Modules: Towards Physical Data Independence in XML Databases. In *XIME-P*, 2005.

[ABM+06] Andrei Arion, Véronique Benzaken, Ioana Manolescu, Yannis Papakonstantinou, and Ravi Vijay. Algebra-Based identification of tree patterns in XQuery. In *FQAS*, 2006.

[ABMP07a] Andrei Arion, Véronique Benzaken, Ioana Manolescu, and Yannis Papakonstantinou. Structured Materialized Views for XML Queries. In *VLDB*, 2007.

[ABMP07b] Andrei Arion, Angela Bonifati, Ioana Manolescu, and Andrea Pugliese. XQueC: A query-conscious compressed XML database. *TOIT*, 2007.

[ACFR02] Serge Abiteboul, Sophie Cluet, Guy Ferran, and Marie-Christine Rousset. The Xyleme project. *Computer Networks*, 2002.

[ACN00] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*, 2000.

[AD98] Serge Abiteboul and Oliver M. Duschka. Complexity of Answering Queries Using Materialized Views. In *PODS*, 1998.

[ADD+11] Yael Amsterdamer, Susan B Davidson, Daniel Deutch, Tova Milo, Julia Stoyanovich, and Val Tannen. Putting lipstick on pig: enabling database-style workflow provenance. *PVLDB*, 2011.

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Communications of the ACM*, 2010.

[AKJP+02] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *ICDE*, 2002.

[AKO08] Parag Agrawal, Daniel Kifer, and Christopher Olston. Scheduling shared scans of large data files. *PVLDB*, 2008.

[AMP+08] Serge Abiteboul, Ioana Manolescu, Neoklis Polyzotis, Nicoleta Preda, and Chong Sun. XML processing in DHT networks. In *ICDE*, 2008.

[Ast] AsterixDB. http://asterixdb.ics.uci.edu/.

[ASU79a] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. Efficient Optimization of a Class of Relational Expressions. *TODS*, 1979.

[ASU79b] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. Equivalences Among Relational Expressions. *SIAM J. Comput.*, 1979.

[AU10] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, 2010.

[AWS] Amazon Web Services. http://aws.amazon.com/.

[Bas] BaseX. http://basex.org/products/xquery/.

[BBC+11] Alexander Behm, Vinayak R. Borkar, Michael J. Carey, Raman Grover, Chen Li, Nicola Onose, Rares Vernica, Alin Deutsch, Yannis Papakonstantinou, and Vassilis J. Tsotras. ASTERIX: Towards a Scalable, Semistructured Data Platform for Evolving-world Models. *Distributed and Parallel Databases*, 2011.

[BC06] Angela Bonifati and Alfredo Cuzzocrea. Storing and retrieving XPath fragments in structured P2P networks. *Data Knowl. Eng.*, 2006.

[BC10] Michael Benedikt and James Cheney. Destabilizers and Independence of XML Updates. *PVLDB*, 2010.

[BCD+11] Philip A. Bernstein, Istvan Cseri, Nishant Dani, Nigel Ellis, Ajay Kalhan, Gopal Kakivaya, David B. Lomet, Ramesh Manne, Lev Novik, and Tomas Talius. Adapting Microsoft SQL server for cloud computing. In *ICDE*, 2011.

[BCG+11] Vinayak R. Borkar, Michael J. Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, 2011.

[BCJ+05] Kevin S. Beyer, Roberta Cochrane, Vanja Josifovski, Jim Kleewein, George Lapis, Guy M. Lohman, Robert Lyle, Fatma Özcan, Hamid Pirahesh, Normen Seemann, Tuong C. Truong, Bert Van der Linden, Brian Vickery, and Chun Zhang. System RX: One Part Relational, One Part XML. In *SIGMOD*, 2005.

[BCM+13] Nicole Bidoit, Dario Colazzo, Noor Malla, Federico Ulliana, Maurizio Nolè, and Carlo Sartiani. Processing XML queries and updates on map/reduce clusters (demo). In *EDBT*, 2013.

[BCRG+14] Francesca Bugiotti, Jesús Camacho-Rodríguez, François Goasdoué, Zoi Kaoudi, Ioana Manolescu, and Stamatis Zampetakis. SPARQL Query Processing in the Cloud. In Andreas Harth, Katja Hose, and Ralf Schenkel, editors, *Linked Data Management*, Emerging Directions in Database Systems and Applications. Chapman and Hall/CRC, April 2014.

[BEG+11] Kevin S. Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Y. Eltabakh, Carl-Christian Kanne, Fatma Özcan, and Eugene J. Shekita. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. *PVLDB*, 2011.

[BEH+10] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. In *SoCC*, 2010.

[BFG+08] Matthias Brantner, Daniela Florescu, David A. Graf, Donald Kossmann, and Tim Kraska. Building a database on S3. In *SIGMOD*, 2008.

[BFG+09] Matthias Brantner, Daniela Florescu, David A. Graf, Donald Kossmann, and Tim Kraska. Building a database in the Cloud. Technical Report, ETH Zürich, 2009.

[BGMS13] Angela Bonifati, Martin Goodfellow, Ioana Manolescu, and Domenica Sileo. Algebraic incremental maintenance of XML views. *TODS*, 2013.

[BGvK+06] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *SIGMOD*, 2006.

[BK90] Catriel Beeri and Yoram Kornatzky. Algebraic optimization of object-oriented query languages. In *ICDT*, 1990.

[BKS02] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD*, 2002.

[BOB+04] Andrey Balmin, Fatma Özcan, Kevin S. Beyer, Roberta J. Cochrane, and Hamid Pirahesh. A framework for using materialized XPath views in XML query processing. In *VLDB*, 2004.

[BPE+10] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A Comparison of Join Algorithms for Log Processing in MapReduce. In *SIGMOD*, 2010.

[BTCU12] Nicole Bidoit-Tollu, Dario Colazzo, and Federico Ulliana. Type-Based Detection of XML Query-Update Independence. *PVLDB*, 2012.

[CDZ06] Yi Chen, Susan B. Davidson, and Yifeng Zheng. An efficient XPath query processor for XML streams. In *ICDE*, 2006.

[CGM11] Federico Cavalieri, Giovanna Guerrini, and Marco Mesiti. Dynamic Reasoning on XML Updates. In *EDBT*, 2011.

[CHS02] Rada Chirkova, Alon Y. Halevy, and Dan Suciu. A Formal Perspective on the View Selection Problem. *VLDB Journal*, 2002.

[CJLP03] Zhimin Chen, H. V. Jagadish, Laks Lakshmanan, and Stelios Paparizos. From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. In *VLDB*, 2003.

[CJMB11] Carlo Curino, Evan P. C. Jones, Samuel Madden, and Hari Balakrishnan. Workload-aware database monitoring and consolidation. In *SIGMOD*, 2011.

[CLK+12] Hyebong Choi, Kyong-Ha Lee, Soo-Hyong Kim, Yoon-Joon Lee, and Bongki Moon. HadoopXML: A suite for parallel processing of massive XML data with multiple twig pattern queries (demo). In *CIKM*, 2012.

[CM77] Ashok K. Chandra and Philip M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *STOC*, 1977.

[CM94] Sophie Cluet and Guido Moerkotte. Classification and optimization of nested queries in object bases. Technical report, 1994.

[CR00] Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. *Theor. Comput. Sci.*, 2000.

[CRCM12] Jesús Camacho-Rodríguez, Dario Colazzo, and Ioana Manolescu. Building Large XML Stores in the Amazon Cloud. In *Data Management in the Cloud (DMC) Workshop*, 2012.

[CRCM13] Jesús Camacho-Rodríguez, Dario Colazzo, and Ioana Manolescu. Web Data Indexing in the Cloud: Efficiency and Cost Reductions. In *EDBT*, 2013.

[CRCM14] Jesús Camacho-Rodríguez, Dario Colazzo, and Ioana Manolescu. PAX-Query: A Massively Parallel XQuery Processor. In *Workshop on Data analytics in the Cloud (DanaC)*, 2014.

[CV93] Surajit Chaudhuri and Moshe Y. Vardi. Optimization of Real Conjunctive Queries. In *PODS*, 1993.

[DFS99] Alin Deutsch, Mary Fernandez, and Dan Suciu. Storing Semistructured Data with STORED. In *SIGMOD*, 1999.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[DGG+86] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. GAMMA - A High Performance Dataflow Database Machine. In *VLDB*, 1986.

[DKA09] Debabrata Dash, Verena Kantere, and Anastasia Ailamaki. An Economic Model for Self-Tuned Cloud Caching. In *ICDE*, 2009.

[DN14] Christos Doulkeridis and Kjetil Nørvåg. A Survey of Large-scale Analytical Query Processing in MapReduce. *VLDB Journal*, 2014.

[DPT99] Alin Deutsch, Lucian Popa, and Val Tannen. Physical Data Independence, Constraints, and Optimization with Universal Plans. In *VLDB*, 1999.

[DPX04] Alin Deutsch, Yannis Papakonstantinou, and Yu Xu. The NEXT Logical Framework for XQuery. In *VLDB*, 2004.

[DTCO03] David DeHaan, David Toman, Mariano P. Consens, and M. Tamer Özsu. A Comprehensive XQuery to SQL Translation Using Dynamic Interval Encoding. In *SIGMOD*, 2003.

[EA12] Iman Elghandour and Ashraf Aboulnaga. ReStore: reusing results of MapReduce jobs. *PVLDB*, 2012.

[EDAA11] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *SIGMOD*, 2011.

[ETKM12] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. Spinning Fast Iterative Data Flows. *PVLDB*, 2012.

[FK99a] Daniela Florescu and Donald Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Eng. Bull.*, 1999.

[FK99b] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[FKT86] Shinya Fushimi, Masaru Kitsuregawa, and Hidehiko Tanaka. An Overview of The System Software of A Parallel Relational Database Machine GRACE. In *VLDB*, 1986.

[FLGP11] Leonidas Fegaras, Chengkai Li, Upa Gupta, and Jijo Philip. XML Query Optimization in Map-Reduce. In *WebDB*, 2011.

[FZRL09] Ian T. Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud Computing and Grid Computing 360-Degree Compared. *CoRR*, abs/0901.0131, 2009.

[GCP] Google Cloud Platform. http://cloud.google.com/.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP*, 2003.

[Gir] Apache Giraph. http://giraph.apache.org/.

[GL95] Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. In *SIGMOD Record*, 1995.

[GL01] Jonathan Goldstein and Per-Åke Larson. Optimizing Queries Using Materialized Views: A practical, scalable solution. In *SIGMOD*, 2001.

[GM93] Stéphane Grumbach and Tova Milo. Towards Tractable Algebras for Bags. In *PODS*, 1993.

[Gra93] Goetz Graefe. The Volcano optimizer generator: Extensibility and efficient search. In *ICDE*, 1993.

[Gup97] Himanshu Gupta. Selection of Views to Materialize in a Data Warehouse. In *ICDT*, 1997.

[Gur] Gurobi Optimizer. http://www.gurobi.com.

[GW97] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.

[GWJD03] Leonidas Galanis, Yuan Wang, Shawn R. Jeffery, and David J. DeWitt. Locating Data Sources in Large Distributed Systems. In *VLDB*, 2003.

[Had] Apache Hadoop. http://hadoop.apache.org/.

[Hal01] Alon Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 2001.

[HB11] Herodotos Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of MapReduce programs. *PVLDB*, 2011.

[HPS+12] Fabian Hueske, Mathias Peters, Matthias J. Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. Opening the Black Boxes in Data Flow Optimization. *PVLDB*, 2012.

[HRL+12] Arvid Heise, Astrid Rheinländer, Marcus Leich, Ulf Leser, and Felix Naumann. Meteor/Sopremo: An Extensible Query Language and Operator Model . In *BIGDATA*, 2012.

[HRU96] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing Data Cubes Efficiently. In *SIGMOD*, 1996.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys*, 2007.

[ISO03] Information technology – Database languages – SQL – Part 14: XML-Related Specifications (SQL/XML). Norm ISO/IEC 9075-14, 2003.

[Jar85] Matthias Jarke. Common Subexpression Isolation in Multiple Query Optimization. In *Query Processing in Database Systems*, 1985.

[JCR11] Eaman Jahani, Michael J. Cafarella, and Christopher Ré. Automatic Optimization for MapReduce Programs. *PVLDB*, 2011.

[Jen] Apache Jena. http://jena.apache.org/.

[JK83] David S. Johnson and Anthony C. Klug. Optimizing Conjunctive Queries that Contain Untyped Variables. *SIAM J. Comput.*, 1983.

[JTC11] David Jiang, Anthony K. H. Tung, and Gang Chen. MAP-JOIN-REDUCE: Toward Scalable and Efficient Data Analysis on Large Clusters. *TKDE*, 2011.

[KBNK02] Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Henry F. Korth. Covering indexes for branching path queries. In *SIGMOD*, 2002.

[KCS11] Shahan Khatchadourian, Mariano P. Consens, and Jérôme Siméon. Having a ChuQL at XML on the Cloud. In *AMW*, 2011.

[KDF+11] Verena Kantere, Debabrata Dash, Grégory François, Sofia Kyriakopoulou, and Anastasia Ailamaki. Optimal Service Pricing for a Cloud Cache. *TKDE*, 2011.

[KDGA11] Verena Kantere, Debabrata Dash, Georgios Gratsias, and Anastasia Ailamaki. Predicting cost amortization for query services. In *SIGMOD*, 2011.

[KHAK09] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency Rationing in the Cloud: Pay Only when It Matters. *PVLDB*, 2009.

[KKL10] Donald Kossmann, Tim Kraska, and Simon Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *SIGMOD*, 2010.

[KKM13] Konstantinos Karanasos, Asterios Katsifodimos, and Ioana Manolescu. Delta: Scalable Data Dissemination under Capacity Constraints. *PVLDB*, 2013.

[KKMZ11] Konstantinos Karanasos, Asterios Katsifodimos, Ioana Manolescu, and Spyros Zoupanos. The ViP2P Platform: XML Views in P2P. *CoRR*, arXiv:1112.2610, 2011.

[KKMZ12] Konstantinos Karanasos, Asterios Katsifodimos, Ioana Manolescu, and Spyros Zoupanos. ViP2P: Efficient XML Management in DHT Networks. In *ICWE*, 2012.

[KMV12] Asterios Katsifodimos, Ioana Manolescu, and Vasilis Vassalos. Materialized View Selection for XQuery Workloads. In *SIGMOD*, 2012.

[KOD10] Patrick Kling, M. Tamer Özsu, and Khuzaima Daudjee. Generating Efficient Execution Plans for Vertically Partitioned XML Databases. *PVLDB*, 2010.

[KP05] Georgia Koloniari and Evaggelia Pitoura. Peer-to-peer management of XML data: issues and research challenges. *SIGMOD Record*, 2005.

[KSTI11] Herald Kllapi, Eva Sitaridi, Manolis M. Tsangaris, and Yannis E. Ioannidis. Schedule optimization for data processing flows on the cloud. In *SIGMOD*, 2011.

[LMS05] Paul J. Leach, Michael Mealling, and Rich Salz. A Universally Unique IDentifier (UUID) URN Namespace. IETF RFC 4122, July 2005.

[LMSS95] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering Queries Using Views. In *PODS*, 1995.

[LOOW14] Feng Li, Beng Chin Ooi, M. Tamer Özsu, and Sai Wu. Distributed Data Management Using MapReduce. *ACM Comput. Surv.*, 2014.

[LSH+14] Jeff LeFevre, Jagan Sankaranarayanan, Hakan Hacigumus, Junichi Tatemura, Neoklis Polyzotis, and Michael J. Carey. Opportunistic Physical Design for Big Data Analytics. In *SIGMOD*, 2014.

[LW97] Leonid Libkin and Limsoon Wong. Query Languages for Bags and Aggregate Functions. *Journal of Computer and System Sciences*, 1997.

[MFK01] Ioana Manolescu, Daniela Florescu, and Donald Kossmann. Answering XML Queries on Heterogeneous Data Sources. In *VLDB*, 2001.

[MHM06] Norman May, Sven Helmer, and Guido Moerkotte. Strategies for query unnesting in XML databases. *TODS*, 2006.

[MKVZ11] Ioana Manolescu, Konstantinos Karanasos, Vasilis Vassalos, and Spyros Zoupanos. Efficient XQuery Rewriting using Multiple Views. In *ICDE*, 2011.

[MLK+05] Ravi Murthy, Zhen Hua Liu, Muralidhar Krishnaprasad, Sivasankaran Chandrasekar, Anh-Tuan Tran, Eric Sedlar, Daniela Florescu, Susan Kotsovolos, Nipun Agarwal, Vikas Arora, and Viswanathan Krishnamurthy. Towards an Enterprise XML Architecture. In *SIGMOD*, 2005.

[MMS07] Philippe Michiels, George A. Mihaila, and Jérôme Siméon. Put a Tree Pattern in Your Algebra. In *ICDE*, 2007.

[MP05] Ioana Manolescu and Yannis Papakonstantinou. XQuery Midflight: Emerging Database-Oriented Paradigms and a Classification of Research Advances. In *ICDE*, 2005.

[MPV09] Ioana Manolescu, Yannis Papakonstantinou, and Vasilis Vassalos. XML Tuple Algebra. In *Encyclopedia of Database Systems*. 2009.

[MS99] Tova Milo and Dan Suciu. Index structures for path expressions. In *ICDT*, 1999.

[MS02] Gerome Miklau and Dan Suciu. Containment and equivalence for an XPath fragment. In *PODS*, 2002.

[MS05] Bhushan Mandhani and Dan Suciu. Query caching and view selection for XML databases. In *VLDB*, 2005.

[NPM+10] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. MRShare: sharing across multiple queries in MapReduce. *PVLDB*, 2010.

[ODPC06] Nicola Onose, Alin Deutsch, Yannis Papakonstantinou, and Emiran Curtmola. Rewriting nested XML queries using nested views. In *SIGMOD*, 2006.

[OR11] Alper Okcan and Mirek Riedewald. Processing theta-joins using MapReduce. In *SIGMOD*, 2011.

[ORS+08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*, 2008.

[OV11] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer Publishing Company, Incorporated, 3rd edition, 2011.

[PCS+04] Shankar Pal, Istvan Cseri, Oliver Seeliger, Gideon Schaller, Leo Giakoumakis, and Vasili Zolotov. Indexing XML Data Stored in a Relational Database. In *VLDB*, 2004.

[PH01] Rachel Pottinger and Alon Y. Halevy. MiniCon: A scalable algorithm for answering queries using views. *VLDB Journal*, 2001.

[Piga] Apache Pig. http://pig.apache.org/.

[Pigb] PigMix. http://cwiki.apache.org/confluence/display/PIG/PigMix.

[PS88] Jooseok Park and Arie Segev. Using Common Subexpressions to Optimize Multiple Queries. In *ICDE*, 1988.

[PS96] Alexandra Poulovassilis and Carol Small. Algebraic query optimisation for database programming languages. *VLDB Journal*, 1996.

[PWLJ04] Stelios Paparizos, Yuqing Wu, Laks V. S. Lakshmanan, and H. V. Jagadish. Tree Logical Classes for Efficient Evaluation of XQuery. In *SIGMOD*, 2004.

[Qiz] Qizx/open. http://www.axyana.com/qizxopen/.

[RG03] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2003.

[RSF06] Christopher Re, Jérôme Siméon, and Mary F. Fernández. A Complete and Efficient Algebraic Compiler for XQuery. In *ICDE*, 2006.

[RSSB00] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. In *SIGMOD*, 2000.

[Sax] Saxon. http://www.saxonica.com/.

[SDQR10] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *PVLDB*, 2010.

[Sel88] Timos K. Sellis. Multiple-query optimization. *TODS*, 1988.

[SETM13] Sebastian Schelter, Stephan Ewen, Kostas Tzoumas, and Volker Markl. "All roads lead to Rome": optimistic recovery for distributed iterative data processing. In *CIKM*, 2013.

[SLZ12] Yasin N. Silva, Paul-Ake Larson, and Jingren Zhou. Exploiting Common Subexpressions for Cloud Query Processing. In *ICDE*, 2012.

[Sol79] Martin K. Solomon. Some Properties of Relational Expressions. In *Proc. of the 17th Annual Southeast Regional Conference*, 1979.

[Spa] Apache Spark. http://spark.apache.org/.

[Sto] Apache Storm. http://storm.incubator.apache.org/.

[Str] Stratosphere System. http://stratosphere.eu/.

[SWK+02] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A benchmark for XML data management. In *VLDB*, 2002.

[SZ10] Raffael Stein and Valentin Zacharias. RDF On Cloud Number Nine. In *NeFoRS Workshop*, 2010.

[TK14] Immanuel Trummer and Christoph Koch. Approximation Schemes for Many-objective Query Optimization. In *SIGMOD*, 2014.

[TS97] Dimitri Theodoratos and Timos K. Sellis. Data Warehouse Configuration. In *VLDB*, 1997.

[TSJ+10] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. Hive - a PB scale data warehouse using Hadoop. In *ICDE*, 2010.

[TVB+02] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *SIGMOD*, 2002.

[TYÖ+08] Nan Tang, Jeffrey Xu Yu, M. Tamer Özsu, Byron Choi, and Kam-Fai Wong. Multiple Materialized View Selection for XPath Query Rewriting. In *ICDE*, 2008.

[TYT+09] Nan Tang, Jeffrey Xu Yu, Hao Tang, M. Tamer Özsu, and Peter A. Boncz. Materialized view selection in XML databases. In *DASFAA*, 2009.

[VCL10] Rares Vernica, Michael J. Carey, and Chen Li. Efficient Parallel Set-similarity Joins Using MapReduce. In *SIGMOD*, 2010.

[VMD+13] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SoCC*, 2013.

[VXQ] Apache VXQuery. http://incubator.apache.org/vxquery/.

[W3C08] W3C. Extensible Markup Language (XML) 1.0. http://www.w3.org/TR/REC-xml/, November 2008.

[W3C14a] W3C. XML Path Language (XPath) 3.0. http://www.w3.org/TR/xpath-30/, April 2014.

[W3C14b] W3C. XQuery 3.0: An XML Query Language. http://www.w3.org/TR/xquery-30/, April 2014.

[W3C14c] W3C. XQuery and XPath Data Model 3.0. http://www.w3.org/TR/xpath-datamodel-30/, April 2014.

[WA] Windows Azure. http://www.windowsazure.com/.

[WC13] Guoping Wang and Chee-Yong Chan. Multi-Query Optimization in MapReduce Framework. *PVLDB*, 2013.

[WK09] Daniel Warneke and Odej Kao. Nephele: efficient parallel data processing in the cloud. In *SC-MTAGS*, 2009.

[YDHP07] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: Simplified Relational Data Processing on Large Clusters. In *SIGMOD*, 2007.

[YKL97] Jian Yang, Kamalakar Karlapalem, and Qing Li. Algorithms for Materialized View Design in Data Warehousing Environment. In *VLDB*, 1997.

[YL87] H. Z. Yang and Per-Åke Larson. Query Transformation for PSJ-Queries. In *VLDB*, 1987.

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *HotCloud*, 2010.

[ZLFL07] Jingren Zhou, Per-Åke Larson, Johann Christoph Freytag, and Wolfgang Lehner. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD*, 2007.

[ZPR02] Xin Zhang, Bradford Pielech, and Elke A. Rundesnteiner. Honey, I shrunk the XQuery!: an XML algebra optimization approach. In *WIDM*, 2002.

Appendix A

PAXQuery Experimental Queries

This section lists the XQuery queries used in the experimental section of Chapter 4.

Query 1. *Return the name of the person with ID 'person0'.*

```
let $pc := collection('XMarkPeople')
for $p in $pc/site/people/person[@id="person0"]
let $n := $p/name/text()
return $n
```

Query 2. *List the names of items registered in Australia along with their descriptions.*

```
let $ic := collection('XMarkItems')
for $i in $ic/site/regions/australia/item
let $n := $i/name/text(), $d := $i/description
return <item name="{$n}">{$d}</item>
```

Query 3. *Return the names of all items in Europe whose description contains the word 'gold'.*

```
let $ic := collection('XMarkItems')
for $i in $ic/site//europe/item, $d in $i/description/text/text()
let $n := $i/name/text()
where contains($d, "gold")
return $n
```

Query 4. *Print the keywords in emphasis in annotations of closed auctions.*

```
let $cc := collection('XMarkClosedAuctions')
for $a in $cc/site/closed_auctions/closed_auction/annotation/description/
            parlist/listitem/parlist/listitem/text/emph/keyword/text()
return <text>{$a}</text>
```

Query 5. *Return the IDs of those auctions that have one or more keywords in emphasis.*

```
let $cc := collection('XMarkClosedAuctions')
for $a in $cc/site/closed_auctions/closed_auction
for $k in $a/annotation/description/parlist/listitem/parlist/listitem/text/
          emph/keyword/text()
let $s := $a/seller/@person
where not(empty($k))
return <person id="{$s}"/>
```

Query 6. *Which persons have a homepage?*

```
let $pc := collection('XMarkPeople')
for $p in $pc/site/people/person
let $h := $p/homepage, $n := $p/name/text()
where not(empty($h))
return <person name="{$n}"/>
```

Query 7. *How many sold items cost more than 40?*

```
let $cc := collection('XMarkClosedAuctions')
let $p :=
  for $i in $cc/site/closed_auctions/closed_auction
  where $i/price/text() >= 40
  return $i/price
let $c := count($p)
return $c
```

Query 8. *How many items are listed on all continents?*

```
let $ic := collection('XMarkItems')
let $i := $ic/site/regions//item
let $c := count($i)
return $c
```

Query 9. *List the number of buyers per city of France.*

```
let $pc := collection('XMarkPeople'),
    $cc := collection('XMarkClosedAuctions')
for $p in $pc/site/people/person[address/country/text()='France']
let $a := $p/address/city/text()
for $c in $cc/site/closed_auctions/closed_auction, $i in $p/@id,
          $b in $c/buyer/@person
where $i = $b
group by $a
return <res><city>{$a}</city><num>{count($p)}</num></res>
```

Query 10. *List the names of persons and the names of the items they bought in Europe.*

```
let $pc := collection('XMarkPeople'),
    $cc := collection('XMarkClosedAuctions'),
    $ic := collection('XMarkItems')
let $ca := $cc/site/closed_auctions/closed_auction,
    $ei := $ic/site/regions/europe/item
for $p in $pc/site/people/person
let $pn := $p/name/text()
let $a :=
  for $t in $ca, $i in $p/@id, $b in $t/buyer/@person
  let $n :=
    for $t2 in $ei, $ti2 in $t2/@id, $ti in $t/itemref/@item
    where $ti = $ti2
    return $t2
  let $in :=
    for $it in $n/name/text()
    return <item>{$it}</item>
  where $i = $b
  return $in
return <person name="{$pn}">{$a}</person>
```

Query 11. *List all persons according to their interest; use French markup in the result.*

```
let $pc := collection('XMarkPeople')
for $i in distinct-values($pc/site/people/person/profile/interest/@category)
let $p :=
  for $t in $pc/site/people/person, $c in $t/profile/interest/@category
  let $r1 := $t/profile/gender/text(), $r2 := $t/profile/age/text(),
```

```
      $r3 := $t/profile/education/text(), $r4 := $t/profile/@income,
      $r5 := $t/name/text(), $r6 := $t/address/street/text(),
      $r7 := $t/address/city/text(), $r8 := $t/address/country/text(),
      $r9 := $t/emailaddress/text(), $r10 := $t/homepage/text(),
      $r11 := $t/creditcard/text()
  where $c = $i
  return
  <personne>
    <statistiques>
      <sexe>{$r1}</sexe><age>{$r2}</age>
      <education>{$r3}</education><revenu>{$r4}</revenu>
    </statistiques>
    <coordonnees>
      <nom>{$r5}</nom><rue>{$r6}</rue>
      <ville>{$r7}</ville><pays>{$r8}</pays>
      <reseau>
        <courrier>{$r9}</courrier><pagePerso>{$r10}</pagePerso>
      </reseau>
    </coordonnees>
    <cartePaiement>{$r11}</cartePaiement>
  </personne>
return <categorie><id>{$i}</id>{$p}</categorie>
```

Query 12. *List the names of persons and the number of items they bought.*

```
let $pc := collection('XMarkPeople'),
    $cc := collection('XMarkClosedAuctions')
for $p in $pc/site/people/person
let $n := $p/name/text()
let $a :=
  for $t in $cc/site/closed_auctions/closed_auction, $b in $t/buyer/@person,
      $i in $p/@id
  where $b = $i
  return $t
let $c := count($a)
return <item person="{$n}">{$c}</item>
```

Query 13. *List the name of users in France and the items that they bought or sold in an auction.*

```
let $pc := collection('XMarkPeople'),
    $cc := collection('XMarkClosedAuctions')
for $p in $pc/site/people/person, $i in $p/@id,
    $ad in $p/address/country/text()
let $a :=
  for $c in $cc//closed_auction, $b in $c/buyer/@person,
      $s in $c/seller/@person
  let $ir := $c/itemref
  where $i = $b or $i = $s
  return $ir
let $n := $p/name
where $ad = 'France'
return <res>{$n,$a}</res>
```

Query 14. *For each rich person, list the number of cars-related items currently on sale whose price does not exceed* 0.02% *of the person's income.*

```
let $pc := collection('XMarkPeople'),
    $oc := collection('XMarkOpenAuctions')
for $p in $pc/site/people/person
let $l :=
  for $o in $oc/site/open_auctions/open_auction, $i in $o/initial/text(),
      $si in $p/profile/@income, $a in $o/annotation//text/text()
  let $x := 5000*$i
```

```
  where $si > $x and contains($a,"car")
  return $i
for $li in $p/profile/@income
let $n := count($l)
where $li > 200000
return <items person="{$li}">{$n}</items>
```

Appendix B

Algebra Equivalences in PigReuse

This section enumerates the different laws that PigReuse (see Chapter 5) is capable of applying to detect algebra expressions equivalences, which have been extensively studied previously [BK90, GL95, PS96, RG03].

Equivalence 1. *Cascading of selections:*

$$\sigma\langle p_1\rangle(\sigma\langle p_2\rangle(\ldots(\sigma\langle p_n\rangle(\underline{\mathsf{var}}))\ldots)) \equiv \sigma\langle p_1 \wedge p_2 \wedge \ldots \wedge p_n\rangle(\underline{\mathsf{var}})$$

Equivalence 2. *Commutativity of selection:*

$$\sigma\langle p_1\rangle(\sigma\langle p_2\rangle(\underline{\mathsf{var}})) \equiv \sigma\langle p_2\rangle(\sigma\langle p_1\rangle(\underline{\mathsf{var}}))$$

Equivalence 3. *Cascading of projections:*

$$\pi\langle C_1\rangle(\pi\langle C_2\rangle(\ldots(\pi\langle C_n\rangle(\underline{\mathsf{var}}))\ldots)) \equiv \pi\langle C_1\rangle(\underline{\mathsf{var}})$$

where C_i is a set of columns such that $C_i \subseteq C_{i+1}$, $\forall i = 1, \ldots, n-1$.

Equivalence 4. *Cascading of additive union:*

$$\underline{\mathsf{var}}_1 \uplus (\underline{\mathsf{var}}_2 \uplus (\ldots \uplus (\underline{\mathsf{var}}_{n-1} \uplus \underline{\mathsf{var}}_n)\ldots)) \equiv \underline{\mathsf{var}}_1 \uplus \underline{\mathsf{var}}_2 \uplus \ldots \uplus \underline{\mathsf{var}}_n$$

Equivalence 5. *Commutativity of additive union:*

$$\underline{\mathsf{var}}_1 \uplus \underline{\mathsf{var}}_2 \equiv \underline{\mathsf{var}}_2 \uplus \underline{\mathsf{var}}_1$$

Equivalence 6. *Associativity of additive union:*

$$\underline{\mathsf{var}}_1 \uplus (\underline{\mathsf{var}}_2 \uplus \underline{\mathsf{var}}_3) \equiv (\underline{\mathsf{var}}_1 \uplus \underline{\mathsf{var}}_2) \uplus \underline{\mathsf{var}}_3$$

Equivalence 7. *Cascading of cross:*

$$\underline{\mathsf{var}}_1 \times (\underline{\mathsf{var}}_2 \times (\ldots \times (\underline{\mathsf{var}}_{n-1} \times \underline{\mathsf{var}}_n)\ldots)) \equiv \underline{\mathsf{var}}_1 \times \underline{\mathsf{var}}_2 \times \ldots \times \underline{\mathsf{var}}_n$$

Equivalence 8. *Commutativity of cross:*

$$\underline{\mathsf{var}}_1 \times \underline{\mathsf{var}}_2 \equiv\ \underline{\mathsf{var}}_2 \times \underline{\mathsf{var}}_1$$

Equivalence 9. *Associativity of cross:*

$$(\underline{\mathsf{var}}_1 \times \underline{\mathsf{var}}_2) \times \underline{\mathsf{var}}_3 \equiv (\underline{\mathsf{var}}_1 \times \underline{\mathsf{var}}_3) \times \underline{\mathsf{var}}_2$$

Equivalence 10. *Commutativity of cogroup:*

$$\mathit{cogroup}\langle a_1, a_2\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2) \equiv \ \mathit{cogroup}\langle a_2, a_1\rangle(\underline{\mathsf{var}}_2, \underline{\mathsf{var}}_1)$$

Equivalence 11. *Cascading of inner join:*

$$\bowtie\langle a_1, a_2\rangle(\underline{\mathsf{var}}_1, \bowtie\langle a_2, a_3\rangle(\underline{\mathsf{var}}_2, \ldots, \bowtie\langle a_{n-1}, a_n\rangle(\underline{\mathsf{var}}_{n-1}, \underline{\mathsf{var}}_n)\ldots)) \equiv \\ \bowtie\langle a_1, a_2, \ldots, a_n\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2, \ldots, \underline{\mathsf{var}}_n)$$

Equivalence 12. *Commutativity of inner join:*

$$\bowtie\langle a_1, a_2\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2) \equiv \ \bowtie\langle a_2, a_1\rangle(\underline{\mathsf{var}}_2, \underline{\mathsf{var}}_1)$$

Equivalence 13. *Associativity of inner join:*

$$\bowtie\langle a_1, a_2\rangle(\underline{\mathsf{var}}_1, \bowtie\langle a_2, a_3\rangle(\underline{\mathsf{var}}_2, \underline{\mathsf{var}}_3)) \equiv \bowtie\langle a_2, a_3\rangle(\bowtie\langle a_1, a_2\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2), \underline{\mathsf{var}}_3)$$

Equivalence 14. *Cascading of full outer join:*

$$⟗\langle a_1, a_2\rangle(\underline{\mathsf{var}}_1, ⟗\langle a_2, a_3\rangle(\underline{\mathsf{var}}_2, \ldots, ⟗\langle a_{n-1}, a_n\rangle(\underline{\mathsf{var}}_{n-1}, \underline{\mathsf{var}}_n)\ldots)) \equiv \\ ⟗\langle a_1, a_2, \ldots, a_n\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2, \ldots, \underline{\mathsf{var}}_n)$$

Equivalence 15. *Commutativity of full outer join:*

$$⟗\langle a_1, a_2\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2) \equiv ⟗\langle a_2, a_1\rangle(\underline{\mathsf{var}}_2, \underline{\mathsf{var}}_1)$$

Equivalence 16. *Associativity of full outer join:*

$$⟗\langle a_1, a_2\rangle(\underline{\mathsf{var}}_1, ⟗\langle a_2, a_3\rangle(\underline{\mathsf{var}}_2, \underline{\mathsf{var}}_3)) \equiv ⟗\langle a_2, a_3\rangle(⟗\langle a_1, a_2\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2), \underline{\mathsf{var}}_3)$$

Equivalence 17. *Commutativity of selection and projection:*

$$\sigma\langle p\rangle(\pi\langle a_1, \ldots, a_n\rangle(\underline{\mathsf{var}})) \equiv \pi\langle a_1, \ldots, a_n\rangle(\sigma\langle p\rangle(\underline{\mathsf{var}}))$$

where every attribute mentioned in p must be included in $a_1, \ldots, a_n$.

Equivalence 18. *Commutativity of selection and cross:*

$$\sigma\langle p\rangle(\underline{\mathsf{var}}_1 \times \underline{\mathsf{var}}_2) \equiv \sigma\langle p\rangle(\underline{\mathsf{var}}_1) \times \underline{\mathsf{var}}_2$$

where all attributes in p belong to $\underline{\mathsf{var}}_1$. In general, a selection can be replaced by a cascade of selections, and then some of the resulting selections might commute with the cross operator.

Equivalence 19. *Commutativity of selection and inner join:*

$$\sigma\langle p\rangle(\bowtie\langle a_1, a_2\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2)) \equiv \ \bowtie\langle a_1, a_2\rangle(\sigma\langle p\rangle(\underline{\mathsf{var}}_1), \underline{\mathsf{var}}_2)$$

where all attributes in p belong to $\underline{\mathsf{var}}_1$. In general, a selection can be replaced by a cascade of selections, and then some of the resulting selections might commute with the join operator.

Equivalence 20. *Commutativity of selection and left outer join:*

$$\sigma\langle p\rangle(⟕\langle a_1, a_2\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2)) \equiv ⟕\langle a_1, a_2\rangle(\sigma\langle p\rangle(\underline{\mathsf{var}}_1), \underline{\mathsf{var}}_2)$$

where all attributes in p belong to $\underline{\mathsf{var}}_1$. A selection can only be pushed to the left input of a left outer join operator.

Equivalence 21. *Commutativity of selection and right outer join:*

$$\sigma\langle p\rangle(⟖\langle a_1, a_2\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2)) \equiv ⟖\langle a_1, a_2\rangle(\underline{\mathsf{var}}_1, \sigma\langle p\rangle(\underline{\mathsf{var}}_2))$$

where all attributes in p belong to $\underline{\mathsf{var}}_2$. A selection can only be pushed to the right input of a right outer join operator.

Equivalence 22. *Commutativity of projection and inner join:*

$$\pi\langle a_1, \ldots, a_i, a_{i+1}, \ldots, a_n\rangle(⋈\langle a_x, a_y\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2)) \equiv$$
$$⋈\langle a_x, a_y\rangle(\pi\langle a_1, \ldots, a_i\rangle(\underline{\mathsf{var}}_1), \pi\langle a_{i+1}, \ldots, a_n\rangle(\underline{\mathsf{var}}_2))$$

where attributes $a_1, \ldots, a_i$ belong to $\underline{\mathsf{var}}_1$, while attributes $a_{i+1}, \ldots, a_n$ belong to $\underline{\mathsf{var}}_2$. Note that the attributes a_x, a_y must be contained in $a_1, \ldots, a_i, a_{i+1}, \ldots, a_n$.

Equivalence 23. *Commutativity of projection and left outer join:*

$$\pi\langle a_1, \ldots, a_i, a_{i+1}, \ldots, a_n\rangle(⟕\langle a_x, a_y\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2)) \equiv$$
$$⟕\langle a_x, a_y\rangle(\pi\langle a_1, \ldots, a_i\rangle(\underline{\mathsf{var}}_1), \pi\langle a_{i+1}, \ldots, a_n\rangle(\underline{\mathsf{var}}_2))$$

where attributes $a_1, \ldots, a_i$ belong to $\underline{\mathsf{var}}_1$, while attributes $a_{i+1}, \ldots, a_n$ belong to $\underline{\mathsf{var}}_2$. Note that the attributes a_x, a_y must be contained in $a_1, \ldots, a_i, a_{i+1}, \ldots, a_n$.

Equivalence 24. *Commutativity of projection and right outer join:*

$$\pi\langle a_1, \ldots, a_i, a_{i+1}, \ldots, a_n\rangle(⟖\langle a_x, a_y\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2)) \equiv$$
$$⟖\langle a_x, a_y\rangle(\pi\langle a_1, \ldots, a_i\rangle(\underline{\mathsf{var}}_1), \pi\langle a_{i+1}, \ldots, a_n\rangle(\underline{\mathsf{var}}_2))$$

where attributes $a_1, \ldots, a_i$ belong to $\underline{\mathsf{var}}_1$, while attributes $a_{i+1}, \ldots, a_n$ belong to $\underline{\mathsf{var}}_2$. Note that the attributes a_x, a_y must be contained in $a_1, \ldots, a_i, a_{i+1}, \ldots, a_n$.

Equivalence 25. *Commutativity of projection and full outer join:*

$$\pi\langle a_1, \ldots, a_i, a_{i+1}, \ldots, a_n\rangle(⟗\langle a_x, a_y\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2)) \equiv$$
$$⟗\langle a_x, a_y\rangle(\pi\langle a_1, \ldots, a_i\rangle(\underline{\mathsf{var}}_1), \pi\langle a_{i+1}, \ldots, a_n\rangle(\underline{\mathsf{var}}_2))$$

where attributes $a_1, \ldots, a_i$ belong to $\underline{\mathsf{var}}_1$, while attributes $a_{i+1}, \ldots, a_n$ belong to $\underline{\mathsf{var}}_2$. Note that the attributes a_x, a_y must be contained in $a_1, \ldots, a_i, a_{i+1}, \ldots, a_n$.

Appendix C

Pig Latin Experimental Script Workloads

This section lists the Pig Latin scripts used in the experimental section of Chapter 5. We used two different workloads in our experiments: workload W_1 consists of scripts 1-12, while workload W_2 consists of the 20 scripts that we introduce below.

Script 1. *l2.pig. Extract the estimated revenue for the pages visited by registered users.*

```
A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr,
    timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, estimated_revenue;
alpha = LOAD 'users' AS (name, phone, address, city, state, zip);
beta = FOREACH alpha GENERATE name;
C = JOIN B BY user, beta BY name;
STORE C INTO 'l2out';
```

Script 2. *l3.pig. Extract the total estimated revenue per registered user.*

```
A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr,
    timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, estimated_revenue;
alpha = LOAD 'users' AS (name, phone, address, city, state, zip);
beta = FOREACH alpha GENERATE name;
C = JOIN B BY user, beta BY name;
D = GROUP C BY user;
E = FOREACH D GENERATE group, SUM(C.estimated_revenue);
STORE E INTO 'l3out';
```

Script 3. *l4.pig. How many different actions has each registered user done?*

```
A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr,
    timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, action;
C = GROUP B BY user ;
D = FOREACH C {
    aleph = B.action;
    beth = DISTINCT aleph;
    GENERATE group, COUNT(beth);
}
STORE D INTO 'l4out';
```

Script 4. *l5.pig. List the page visitors that are not registered users.*

```
A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr,
    timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user;
alpha = LOAD 'users' AS (name, phone, address, city, state, zip);
beta = FOREACH alpha GENERATE name;
C = COGROUP B BY user, beta BY name;
D = FILTER C BY COUNT(beta) == 0;
E = FOREACH D GENERATE group;
STORE E INTO 'l5out';
```

Script 5. *l6.pig. How long did visitors that queried for a certain term stayed in the page?*

```
A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr,
    timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, action, timespent, query_term, ip_addr,
    timestamp;
C = GROUP B BY query_term;
D = FOREACH C GENERATE group, SUM(B.timespent);
STORE D INTO 'l6out';
```

Script 6. *l7.pig. How many visits did each user do during the morning/afternoon?*

```
A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr,
    timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, timestamp;
C = GROUP B BY user;
D = FOREACH C {
    morning = FILTER B BY timestamp < 43200;
    afternoon = FILTER B BY timestamp >= 43200;
    GENERATE group, COUNT(morning), COUNT(afternoon);
}
STORE D INTO 'l7out';
```

Script 7. *l11.pig. List all the users in the dataset (without repetitions).*

```
A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr,
    timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user;
C = DISTINCT B;
alpha = LOAD 'users' AS (name, phone, address, city, state, zip);
beta = FOREACH alpha GENERATE name;
gamma = DISTINCT beta;
D = UNION C, gamma;
E = DISTINCT D;
STORE E INTO 'l11out';
```

Script 8. *l12.pig. Extract the highest revenue page per user, the total timespent in the page, and the number of queries per action.*

```
A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr,
    timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, action, timespent, query_term,
    estimated_revenue;
C = FILTER B BY user IS NOT null;
alpha = FILTER B BY user IS null;
D = FILTER C BY query_term IS NOT null;
aleph = FILTER C BY query_term IS null;
E = GROUP D BY user;
F = FOREACH E GENERATE group, MAX(D.estimated_revenue);
STORE F INTO 'l12out/highest_value_page_per_user';
beta = GROUP alpha BY query_term;
gamma = FOREACH beta GENERATE group, SUM(alpha.timespent);
STORE gamma INTO 'l12out/total_timespent_per_term';
beth = GROUP aleph BY action;
gimel = FOREACH beth GENERATE group, COUNT(aleph);
STORE gimel INTO 'l12out/queries_per_action';
```

Script 9. *l13.pig. List all the page views together with their associated advanced user (if any).*

```
A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr,
    timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, estimated_revenue;
alpha = LOAD 'power_users' AS (pname, pphone, paddress, pcity, pstate, pzip);
beta = FOREACH alpha GENERATE pname, pphone;
C = JOIN B BY user LEFT, beta BY pname;
STORE C INTO 'l13out';
```

Script 10. *l14.pig. Extract the estimated revenue for the pages visited by registered users.*

```
A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr,
    timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, estimated_revenue;
alpha = LOAD 'users' AS (name, phone, address, city, state, zip);
beta = FOREACH alpha GENERATE name;
C = JOIN B BY user, beta BY name;
STORE C INTO 'l14out';
```

Script 11. *l15.pig. Extract the number of different actions, the average spent time, and the generated revenue, per registered user.*

```
A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr,
    timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, action, timespent, estimated_revenue;
C = GROUP B BY user;
D = FOREACH C {
    beth = DISTINCT B.action;
    ts = DISTINCT B.timespent;
    rev = DISTINCT B.estimated_revenue;
    GENERATE group, COUNT(beth), AVG(ts), SUM(rev);
}
STORE D INTO 'l15out';
```

Script 12. *l16.pig. How much revenue did each registered user generate?*

```
A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr,
    timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, estimated_revenue;
C = GROUP B BY user;
D = FOREACH C {
    F = B.estimated_revenue;
    GENERATE group, SUM(F);
}
STORE D INTO 'l16out';
```

Script 13. *e1.pig. List all the registered users together with their associated page views (if any).*

```
A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr,
    timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, estimated_revenue;
alpha = LOAD 'users' AS (name, phone, address, city, state, zip);
beta = FOREACH alpha GENERATE name;
C = JOIN B BY user RIGHT, beta BY name;
STORE C INTO 'e1out';
```

Script 14. *e2.pig. List all the page views and all the registered users, associating them if possible.*

```
A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr,
    timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, estimated_revenue;
alpha = LOAD 'users' AS (name, phone, address, city, state, zip);
beta = FOREACH alpha GENERATE name;
C = JOIN B BY user FULL, beta BY name;
STORE C INTO 'e2out';
```

Script 15. *e3.pig. How many different actions has each registered user done?*

```
A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr,
    timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, action;
C = GROUP B BY user ;
D = FOREACH C {
    aleph = B.action;
    beth = DISTINCT aleph;
    GENERATE group, COUNT(beth);
}
STORE D INTO 'e3out';
```

Script 16. *e4.pig. List the page views per registered user, together with their information as advanced users (if any).*

```
A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr,
    timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user;
alpha = LOAD 'users' AS (name, phone, address, city, state, zip);
beta = FOREACH alpha GENERATE name;
X = LOAD 'power_users' AS (pname, pphone, paddress, pcity, pstate, pzip);
Y = FOREACH X GENERATE pname, pphone;
C = COGROUP B BY user, beta BY name, Y BY pname;
D = FILTER C BY COUNT(beta) == 0;
E = FOREACH D GENERATE group;
STORE E INTO 'e4out';
```

Script 17. *e5.pig. How long did visitors with the same IP address stayed in the page?*

```
A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr,
    timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, action, timespent, query_term, ip_addr,
    timestamp;
C = GROUP B BY ip_addr;
D = FOREACH C GENERATE group, SUM(B.timespent);
STORE D INTO 'e5out';
```

Script 18. *e6.pig. Extract the estimated revenue for the pages visited per registered user.*

```
A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr,
    timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, estimated_revenue;
alpha = LOAD 'users' AS (name, phone, address, city, state, zip);
beta = FOREACH alpha GENERATE name;
C = COGROUP B BY user, beta BY name;
STORE C INTO 'e6out';
```

Script 19. *e7.pig. List all the users in the dataset (without repetitions).*

```
A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr,
    timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user;
C = DISTINCT B;
alpha = LOAD 'users' AS (name, phone, address, city, state, zip);
```

```
beta = FOREACH alpha GENERATE name;
gamma = DISTINCT beta;
D = UNION C, gamma;
E = DISTINCT D;
STORE E INTO 'e7out';
```

Script 20. *e8.pig. Extract the number of different actions, the average spent time, and the generated revenue, per registered user.*

```
A = LOAD 'page_views' AS (user, action, timespent, query_term, ip_addr,
    timestamp, estimated_revenue, page_info, page_links);
B = FOREACH A GENERATE user, action, timespent, estimated_revenue;
C = GROUP B BY user;
D = FOREACH C {
    beth = DISTINCT B.action;
    ts = DISTINCT B.timespent;
    rev = DISTINCT B.estimated_revenue;
    GENERATE group, COUNT(beth), AVG(ts), SUM(rev);
}
STORE D INTO 'e8out';
```