

UNIVERSITÀ DEGLI STUDI DI GENOVA  
FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI  
Corso di Laurea Specialistica in Informatica

# Adaptive query processing for result completeness in the presence of duplicate values

Roald Lengu

Relatori:

**Giovanna Guerrini**

**Marco Mesiti**

Correlatore:

**Francesca Odone**

Anno Accademico 2006 – 2007

# Contents

<b>Introduction</b>	<b>1</b>
<b>1 Data quality</b>	<b>5</b>
1.1 An introduction to data quality	5
1.2 Data quality dimensions	7
1.2.1 Accuracy	7
1.2.2 Completeness	8
1.2.3 Currency	9
1.2.4 Consistency	10
1.3 Object identification	11
1.3.1 Overview	11
1.3.2 Steps of object identification	13
1.3.3 The Fellegi and Sunter method	16
1.4 Offline data cleaning	18
1.4.1 Robust and efficient fuzzy match for data cleaning	18
1.4.2 Efficient set joins on similarity predicates	20
1.4.3 The sjoin operator	21
1.5 Online data cleaning	22
1.5.1 Adaptive execution of variable accuracy functions	23
1.5.2 Declarative support for sensor data cleaning	24
1.5.3 Adaptive cleaning for RFID data streams	25
<b>2 Probability and probability models</b>	<b>27</b>
2.1 Probability basics	27
2.1.1 Sample space and events	27
2.1.2 Notions and axioms of probability	27
2.2 Random variables	28
2.2.1 Definitions	28
2.2.2 Distribution functions	29
2.2.3 Discrete random variables	30
2.2.4 Mean and variance	31
2.2.5 Entropy of a discrete finite probability distribution	32
2.2.6 Some special distributions	32
2.3 Random processes	34
2.4 Hypothesis testing	35
<b>3 Symmetric set hash join algorithm</b>	<b>37</b>
3.1 Pipelined operators	37
3.2 Overview and data structures used	39
3.3 Algorithm	39
3.4 Algorithm analysis	45

3.4.1	Computational analysis . . . . .	45
3.4.2	Space complexity analysis . . . . .	46
3.4.3	Comparison with symmetric hash join . . . . .	47
3.5	Pros and cons . . . . .	49
<b>4</b>	<b>Quality aware adaptive query processing</b>	<b>51</b>
4.1	Initial problem specification . . . . .	51
4.1.1	Online duplicate matching: (Stream) Joins . . . . .	51
4.1.2	General approach to the solution . . . . .	52
4.1.3	Approach validation . . . . .	53
4.2	Some definitions . . . . .	54
4.3	MAR: An adaptive approach to the problem . . . . .	56
4.3.1	General introduction of the MAR architecture . . . . .	56
4.3.2	Pros and cons . . . . .	57
4.4	Problem scenarios . . . . .	58
<b>5</b>	<b>A probability model for the result size estimation</b>	<b>60</b>
5.1	Why do we need a model for our system? . . . . .	60
5.1.1	Detection of missed matches . . . . .	60
5.1.2	A model that describes the result size . . . . .	61
5.2	A binomial model . . . . .	63
5.2.1	Using the binomial model to estimate the result size . . . . .	63
5.2.2	Limitations of the binomial model . . . . .	65
5.3	A hypergeometric model . . . . .	66
5.3.1	Using the hypergeometric model to estimate the result size . . . . .	66
5.3.2	Limitations of the hypergeometric model . . . . .	68
<b>6</b>	<b>A MAR approach to result completeness</b>	<b>70</b>
6.1	Introduction . . . . .	70
6.2	Monitor . . . . .	72
6.3	Analyze . . . . .	74
6.3.1	Exact-approximate join switch . . . . .	74
6.3.2	Approximate-exact join switch . . . . .	75
6.4	Respond . . . . .	76
<b>7</b>	<b>Experimental results</b>	<b>78</b>
7.1	Experimental setting . . . . .	78
7.2	Assumptions . . . . .	81
7.3	Effectiveness of the probability models . . . . .	83
7.4	MAR effectiveness . . . . .	85
	<b>Conclusions</b>	<b>94</b>

# List of Figures

1.1	A <i>Movies</i> relation with data quality problems . . . . .	7
1.2	A <i>Personal</i> relation with null values for the <i>Email</i> attribute . . . . .	9
1.3	Same business seen from three different agencies . . . . .	11
1.4	Object identification steps . . . . .	12
1.5	Notations on matching decision cases . . . . .	16
1.6	The regions of the Fellegi and Sunter model . . . . .	18
1.7	Template for using fuzzy match . . . . .	18
1.8	A set-overlap example . . . . .	21
1.9	ssjoin as a building block operation . . . . .	22
1.10	Basic ssjoin implementation . . . . .	23
1.11	ESP processing stages . . . . .	25
1.12	Fixed smoothing window size filters . . . . .	26
2.1	A random variable X . . . . .	29
2.2	Distribution function graph . . . . .	30
3.1	An algebra expression tree . . . . .	38
3.2	The iterator model . . . . .	38
3.3	The qgram hash table . . . . .	40
3.4	The next method . . . . .	41
3.5	Tables . . . . .	43
3.6	The qgram hash table . . . . .	47
3.7	shjoin-sshjoin costs confront . . . . .	48
4.1	A pipeline architecture example . . . . .	52
4.2	System state machine diagram . . . . .	53
4.3	Horizontal zones in a data stream . . . . .	55
4.4	Problem representation . . . . .	58
5.1	A symmetric hash join scenario . . . . .	62
5.2	A join scenario between <i>Orders</i> and <i>Clients</i> tables . . . . .	64
5.3	A bad sampling example . . . . .	69
6.1	Four self-management aspects . . . . .	71
6.2	The MAPE architecture . . . . .	72
6.3	Result size monitoring . . . . .	73
6.4	Test over exact-approximate switch . . . . .	74
6.5	Test over approximate-exact switch . . . . .	75
6.6	Last 5 <i>perc</i> values . . . . .	75
6.7	Hash table transformation . . . . .	77
7.1	The <i>accidents</i> and <i>location</i> tables . . . . .	78
7.2	The database creation process . . . . .	81

7.3	Effectiveness of the various models . . . . .	85
7.4	The binomial model applied to a no duplicates scenario . . . . .	86
7.5	The hypergeometric model applied to a no duplicates scenario . . . . .	86
7.6	The binomial n-th order distribution model applied to a no duplicates scenario . . . . .	87
7.7	The hypergeometric n-th order distribution model applied to a no duplicates scenario . . . . .	87
7.8	The binomial model applied to a 5% duplicates scenario . . . . .	88
7.9	The hypergeometric model applied to a 5% duplicates scenario . . . . .	88
7.10	The binomial n-th order distribution model applied to a 5% duplicates scenario . . . . .	89
7.11	The hypergeometric n-th order distribution model applied to a 5% duplicates scenario . . . . .	89
7.12	The binomial model applied to a 10% duplicates scenario . . . . .	90
7.13	The hypergeometric model applied to a 10% duplicates scenario . . . . .	90
7.14	The binomial n-th order distribution model applied to a 10% duplicates scenario . . . . .	91
7.15	The hypergeometric n-th order distribution model applied to a 10% duplicates scenario . . . . .	91
7.16	Experiments patterns . . . . .	92
7.17	The effectiveness of the MAR architecture . . . . .	93

# List of Tables

2.1	Distribution function values . . . . .	30
-----	--	----

## Abstract

Secondo J. M. Juran, uno dei fondatori delle *management and quality theories*, i dati sono detti di alta qualità *se possono essere utilizzati in modo appropriato per operations, decision making and planning*. Il termine *Quality of Data* (QoD) quindi, si riferisce ad un aspetto multi-dimensionale che esprime una caratteristica intrinseca dei dati offerti, come opposto all'omologo *Quality of Service*, che si riferisce ad una caratteristica intrinseca di un certo servizio offerto.

Si consideri ad esempio uno scenario in cui vogliamo fare il join di due insiemi di tuple su un attributo del primo che condivide lo stesso dominio con un attributo del secondo. Per esempio, si considerino due insiemi: una lista di fermate di autobus, chiamato *LBT*, in cui ogni fermata viene annotata con l'indirizzo della strada in cui si trova (per esempio diverse linee di Londra, come l'autobus 92, potrebbero avere una fermata annotata con *10 Downing Street*); una lista di attrazioni turistiche, chiamato *LTA*, con i corrispondenti indirizzi (per esempio, *Office and home of the prime minister* potrebbe essere annotata con *10 Downing St*). Si noti che *10 Downing Street* e *10 Downing St* rappresentano lo stesso oggetto reale e sono quindi dei valori mutuamente duplicati. Il termine *duplicato*, si riferisce a delle rappresentazioni simili, ma strettamente diverse, della stessa entità (oggetto) reale, in letteratura chiamati anche duplicati fuzzy [CGM05].

Uno dei rischi della presenza dei duplicati è che la completezza del risultato, un aspetto del QoD, potrebbe non essere raggiunta senza ricorso a misure speciali. Nell'esempio di prima, se l'utente volesse trovare quali autobus potrebbe usare per andare a una qualsiasi attrazione turistica, la coppia (*92, Office and home of the prime minister*) dovrebbe fare parte del risultato, ma in presenza di duplicati questo non potrà succedere. Rispondere in maniera efficace ed efficiente alla presenza dei duplicati è essenziale se si spera che il settore di *data provision* dovrebbe diventare un settore robusto e industrialmente avanzato come quello di *service provision*. Molta ricerca è già stata svolta nell'ambito di come un provider dovrebbe risolvere i problemi causati dalla presenza dei duplicati (vedi survey [BS06, EIV07]), però la maggior parte di questo lavoro consiste in attività di *profiling* e di filtraggio *offline* dei dati, che avvengono in una fase precedente alla generazione dei *data product* finali distribuiti al consumatore.

Da questo punto di vista, una classica contromisura nell'esempio precedente, sarebbe quella di standardizzare le rappresentazioni degli indirizzi. Perciò, prima di integrare i due insiemi, noi potremmo già avere diagnosticato che *LTA* usa delle abbreviazioni, ed invece *LBT* usa le forme complete per rappresentare gli indirizzi. In questo caso, si potrebbe applicare una trasformazione ad *LBT* per farle usare le stesse abbreviazioni di *LTA* risolvendo il problema *offline*. In molti casi però, prendere delle contromisure *offline* potrebbe non essere vantaggioso o possibile. Potrebbe non essere vantaggioso perchè la misura di perturbazione (che è la proporzione dei duplicati nell'insieme) potrebbe essere molto piccola per giustificare una fase preliminare computazionalmente costosa e il conseguente ritardo introdotto nel servizio di *data provision*. Potrebbe, addirittura non essere possibile, per esempio in scenari di *data streaming*, dove al provider non viene data l'opportunità di fare *data profiling* per riconciliare i dati prima della loro consumazione dalla *query* che genera i prodotti finali contrattati dal consumatore. Anche in casi in cui gli input non fossero degli stream, questi potrebbe appartenere ad una terza parte che le mette a disposizione solo sotto richiesta (come per esempio è normale in scenari di *mashup* di integrazione *on-the-fly*) eliminando l'opportunità di disporre di un tempo preliminare per ridurre o eliminare la perturbazione da parte del provider.

Sono quindi poche le situazioni in cui si potrebbero effettuare delle operazioni computazionalmente costose di *data profiling* e *data cleaning offline* da parte dei provider sui dati che distribuiscono ai loro consumatori. Si noti che la probabilità di avere dei duplicati è alta, soprattutto in quegli scenari, come nel nostro esempio, quando non sembra che questi siano dovuti a degli errori, ma ai

risultati di diverse decisioni di design. Nel caso di integrazioni dinamiche *on-the-fly* (come succedrebbe nel nostro caso, se la lista delle fermate e quella delle attrazioni turistiche fossero accedute via *web service* durante una richiesta ad-hoc da parte di un utente di un sito web), non sarebbe possibile eliminare questo rischio con una misura preventiva perchè la scelta degli insiemi e degli attributi da usare è difficile da prevedere nel caso generale.

In questa tesi, proponiamo una classe diversa di contromisure che sono appropriate per contesti dinamici più generici. Noi ci restringiamo al caso in cui i dati sono stati ottenuti come risultato di un join. Dopodiché descriviamo una tecnica di elaborazione di query adattativa (AQP) che permette al provider di confrontare le minacce proferite alla completezza dei dati. La tecnica rileva la presenza inattesa di duplicati nella distribuzione dei valori di un attributo di un insieme che condivide lo stesso dominio con un altro attributo di un altro insieme. La novità più importante del nostro metodo è che questo cerca di applicare delle contromisure durante la creazione di un *data product*, invece di applicarle *offline* come è solito fare, e solo se c'è evidenza che queste servono veramente, invece di farle per *default*, come è solito fare.

In particolare, per garantire la completezza dei risultati in presenza di duplicati, la nostra soluzione usa delle tecniche di rimpiazzamento di operatori in piani di esecuzione pipeline [EFP06] e di join approssimati [CGK06] come segue: l'occorrenza di duplicati in almeno uno degli input del join può causare uno *switch* da join esatto a join approssimato, e possibilmente uno *reverse switch* se i duplicati non vengono più rilevati. A grandi linee la strategia è la seguente

1. usare un join esatto in presenza di duplicati compromette la completezza dei risultati,
2. usare un join approssimato neutralizza la minaccia provocata dalla presenza dei duplicati, ma risulta computazionalmente più costoso.

Nella nostra soluzione, noi monitoriamo l'esecuzione del join e consideriamo la possibilità di fare lo *switch* tra join esatto ed approssimato come risposta all'evidenza che la presenza dei duplicati sta minacciando la completezza del risultato, ma teniamo in considerazione i costi computazionali in modo da tenere basso l'*overhead* di applicare questa contromisura. Quindi, invece di applicare le contromisure come un passo fisso (e quindi pagando un costo computazionale fisso), il nostro metodo fornisce ai provider la possibilità di applicare tali contromisure secondo un approccio di *when-needed* ed *if-required*. Oltre alla sua rilevanza pratica, il nostro contributo illustra la versatilità delle tecniche di AQP [DIR07]. Molto spesso, delle tecniche di AQP sono state usate per garantire degli standard di QoS (soprattutto in piani di esecuzione parallela delle query [DIR07]). Questa tesi dimostra che tecniche di AQP possono essere applicate anche a problemi di QoD, in questo caso, alla completezza del risultato in presenza di duplicati.

I primi due capitoli di questa tesi (1 e 2) danno una panoramica del lavoro già svolto nel settore e introducono alcune nozioni preliminary. Queste nozioni vengono ulteriormente usate nei capitoli successivi (3–7) per spiegare il nostro specifico contributo nell'area.

I principali contributi di questa tesi sono: symmetric set hash join (sshjoin), un nuovo algoritmo approssimato ed incrementale per eseguire operazioni di join in presenza di duplicati; una nuova tecnica di AQP per garantire QoD; una istanziazione dell'approccio generico, in cui una strategia adattativa viene usata per garantire la completezza del risultato; l'adattamento di un insieme di modelli probabilistici per rilevare la presenza di duplicati negli stream di input al join, insieme ad un confronto teorico e sperimentale della loro efficacia; un'analisi costo-beneficio della nostra strategia adattativa tramite un insieme di risultati sperimentali.

# Introduction

According to J. M. Juran, one of the fathers of management and quality theories, data are of high quality *if they are fit for their intended uses in operations, decision making and planning*. The term *Quality of Data* (QoD) though, refers to a multifaceted aspect that expresses an intrinsic characteristic of certain provided data, as opposed to the counterpart *Quality of Service* (QoS), which refers to an intrinsic characteristic of a certain provided service.

Consider a scenario in which we want to join two collections on an attribute of one that shares the same domain with an attribute of the other. For example, consider two collections: one is a bus timetable, call it *LBT*, in which stops are annotated with the street address (e.g., several London routes, such as that of the 92 bus, might have a stop annotated with 10 *Downing Street*), and the other is a list of tourist attractions, call it *LTA*, with their addresses (e.g., in London, the *Office and Home of the British Prime Minister* might be annotated with 10 *Downing St*). Note that 10 *Downing Street* in *LBT* and 10 *Downing St* in *LTA* are mutually duplicate values. By duplicate values (or duplicates, for short) we mean similar, but strictly distinct, representations of the same real world entity. What we call duplicates, others (e.g., [CGM05]) have called fuzzy duplicates.

One impact of duplicates is that an important QoD desideratum, viz., result completeness, may not be met without recourse to special measures. In the example above, if a user wanted to find which bus services he could use to go to which tourist attractions, the tuple (92, *Office and Home of the British Prime Minister*) would be expected in the result, but in the presence of duplicates would not do so. Responding effectively and efficiently to the unexpected presence of duplicates is essential if data services provision is to attain the status of a robust, thriving industrial sector on par with application services provision. While there is a vast literature (e.g., see the surveys [BS06] and [EIV07]) on how providers can deal with duplicates, the bulk of it assumes data profiling and data cleaning activities to take place offline, as a preprocessing step ahead of the generation of data products.

From this viewpoint, a classical countermeasure in the case of the example above would be to standardize the representation of address elements. Thus, in the wake of data profiling, we may have diagnosed that *LTA* uses short forms while *LBT* uses expanded forms when representing addresses. In this case, one possibility is to apply a transformation to *LBT* to make it use the same short forms as *LTA*. Profiling and transformation would, in this case, counteract the kind of threat to result completeness highlighted by our example. However, performing these offline countermeasures may not be advantageous or even feasible. It may not be advantageous because the degree of perturbation (e.g., the proportion of duplicates in a collection) may be too small to make cost-effective the offline step and the necessary lag it introduces in the data provision business process. It may also not be feasible, e.g., in streaming data scenarios, where the provider is simply not given the opportunity to perform offline profiling and reconciliation on inputs prior to their consumption by the query that generates the data product contracted with the consumer. Furthermore, even if the inputs are not streams, they may belong to a third-party and only be made available on demand (e.g., as is typical in mashup-style, on-the-fly integration) thereby precluding the existence of offline time for the provider to reduce or eliminate the perturbations that the inputs may exhibit.

In general, in wide-area settings, where data provision is one of many activities that are carried out by the dynamic orchestration of maximally-decoupled service-oriented context [ACKM03], enforcing

QoD expectations requires more dynamic responses than are normally used in the in-house contexts that providers currently rely on. There are fewer circumstances in which it is cost-effective to perform offline, batch-oriented data profiling and data cleaning on the inputs to data products that a data provider ships to its consumers. We note that the likelihood of duplicates is high, particularly when, as in our example, they seem to be, not errors, but the outcome of different design decisions. In dynamic, on-the-fly integration (as is the case, in our example above, if the bus timetable and the list of tourist attractions are being accessed via web services in the wake of an ad-hoc request by a user of a web site), it is not normally feasible to preempt this risk with a preventive measure because the choice of collections and the choice of attributes to use is difficult to predict in the general case.

In this thesis, we propose a different class of countermeasures which are more appropriate for dynamic, online, wide-area contexts. We constrain ourselves to data products that are specified by a (join) query. We then describe an adaptive query processing (AQP) approach that allows providers to contend with threats to result completeness. The approach specifically contends with the unexpected presence of duplicates in the value distribution of an attribute in one collection whose domain is shared with another attribute in the other collection(s). The main novelty of our approach hinges on applying countermeasures when a data product is being created, rather than offline, as is usual, and only if there is evidence of their need, rather than by default, as is usual.

Specifically, in order to improve result completeness in the presence of duplicates, our solution builds upon adaptive techniques for operator replacement in pipelined plans [EFP06] and on approximate joins [CGK06] as follows: the occurrence of duplicates in at least one of the join inputs can cause a switch from using an exact join algorithm to an approximate one, and possibly a reverse switch if duplicates cease to be detected. Broadly, the rationale for this strategy is that

1. using an exact join algorithm in presence of duplicates compromises result completeness,
2. using an approximate join algorithm neutralizes the threat caused by the presence of duplicates but is more expensive.

In our solution we monitor join execution and we consider switching between exact and approximate join algorithms in response to evidence that the occurrence of duplicates is threatening result completeness, but we take into account costs so as to keep low the overhead of applying countermeasures. Thus, rather than deploying countermeasures in a fixed step (and hence incurring a fixed cost), our contributions empower providers with countermeasures that are applied on a when-needed, if-required basis. Besides its practical relevance, our contributions illustrate the versatility of AQP techniques [DIR07]. Most uses of AQP have been motivated by QoS enforcement (mostly in parallel/distributed query plans [DIR07]). This thesis shows that AQP can also be usefully applied to QoD concerns, in this case, result completeness in the presence of duplicates.

The first two chapters of this thesis (1 and 2) give a panoramic of the work already done in the sector of data quality and introduce some important notions. These notions are further used in the following chapters (3–7) to explain our specific contribution in the area. This thesis is organized as follows:

Chapter 1 introduces the multifaceted notion of data quality in general and stresses its economical importance for all type of enterprises and the issues caused to the economy of powerful states such as the USA from its negligence. The data quality issue is then expressed as a composed concept made of four principal aspects (dimensions), i.e. accuracy, completeness, currency and consistency. In this thesis we focus on the following two aspects: accuracy of a single represented record and completeness of a join result. The object identification problem, as the most investigated of all data quality problems is then introduced in detail, together with its most important steps. Finally some state of the art techniques to solve generally quality and more specifically object identification problems are reported in detail. We build upon these techniques, i.e. [CGK06], to create our symmetric set hash algorithm described in chapter 3.

Chapter 2 gives an overview of probability as a science and describes the technical details of some probability notions and specific distributions used in the remainder of this thesis by our adaptive approach. First of all, the importance of probability as a science is reported together with the most important axioms over which it is built. Random variables are then described as functions representing outcomes of aleatory experiments. Their purpose is that of introducing easier ways to calculate probability of these experiments outcomes. The properties of the binomial and hypergeometric distributions are reported. We further use these distributions together with random processes and hypothesis testing as powerful decision making tools for our adaptive approach.

Chapter 3 introduces a new incremental algorithm for performing approximate joins built upon the former *ssjoin* (symmetric set join) algorithm [CGK06]. The algorithm is incremental as it can be perfectly used in a pipeline architecture, reading input tuples one at a time from its feeding pipelined operators and providing output tuples, i.e. join results, one at a time to its consuming operators in the same pipeline.

The notion of pipelined architectures is then introduced together with the possibility of interrupting the workflow during some special execution points, called *quiescent* points, changing operators used in the pipeline with new ones, continuing with the execution from the interruption point without having to recompute the previously computed results and still guarantee the soundness of the final result. This approach was used in [EFP06] to meet certain QoS standards, such as response time. In the following of this thesis we use the same approach to meet certain QoD standards, such as result completeness. The data structures used by the algorithm are first introduced and further the pseudo-code of the algorithm is reported. After this, a thorough analysis of the computational and spatial complexity of the algorithm is performed. Finally we show its advantages and disadvantages.

Chapter 4 deals with the definition of the specific problem we solve in this thesis as part of the bigger object identification one. We make a short description of the more general problem, and then give a high level description of our subproblem. An adaptive approach is then proposed as a solution to the problem, inspired by an IBM previous paper for autonomic computing [KC03], together with a general way to validate its effectiveness and efficiency. Some important definitions used in the remainder of this thesis are then introduced. Finally the particular subproblem we solve in this thesis is placed as part of the more general object identification problem.

Chapter 5 introduces the importance of having a probability model that estimates the probability of missing matches during a join execution, in order to properly and in time respond by using an approximate join rather than an exact one. We use the probability notions and random variables described in Chapter 2 to first explain the importance of this model for our adaptive component and then tailor two existing well-known models, such as the binomial and hypergeometric one, for our needs. Examples are shown for both models in order to better understand their use in our system and a theoretical comparison of their effectiveness, i.e. the degree of reliability in representing the missing matches event, is made.

Chapter 6 introduces the adaptive component of our system, that pilots the behavior of the managed element, i.e. join algorithm used, as a functional decomposition of three subcomponents

**Monitor** the managed component actual performance and behavior in order to obtain valuable statistics and other type of information,

**Analyze** the previously collected statistics and information to detect problems and/or opportunities,

**Respond** with a new behavior for the managed component in order to solve current problems or exploit in the best way the new opportunities.

Chapter 7 reports an experimental validation of our adaptive system by using synthetic data, but with characteristics similar to real ones. We first describe the way large amounts of synthetic data are generated and further errors are injected in them to simulate a low quality data scenario. We use two data tables, namely *accidents* and *locations*, the first having a foreign key constraint on the second

to perform our adaptive exact-approximate methodology. The various technologies used, such as databases and hardware configuration are then reported in detail. Further, certain assumptions are made on the data, in order to limit ourselves to the testing of the particular problem we are solving, introduced in chapter 4. Several graphics showing experimental results are reported, first to make an experimental comparison of the models introduced in chapter 5, which were only theoretically compared, and second to evaluate the effectiveness of the model with data having different error patterns. In the first case we see that experimental results validate the previously theoretical drawn ones. In the second we see that the effectiveness of the system is relatively high and has the following characteristics

- the system uses only exact join if no duplicates are present in the data, thus avoiding the expensive approach of performing an entire approximate join,
- the system obtains an effectiveness as high as 85% in its best cases and as low as 65% in its worst ones.

with the effectiveness percentage measuring the number of approximate matches actually recovered by the algorithm with respect to a total of 100% recovered by an entirely executed approximate join algorithm.

# Chapter 1

## Data quality

Data quality is becoming more and more an important topic, as enterprises do not only use local data, i.e. the ones contained in their own databases or data warehouses, but large quantities of data with heterogeneous characteristics and being delivered from different providers have to be integrated. The quality issue in these cases is essential for an appropriate and easy to do integration. In this chapter the most important notions of data quality are discussed as well as some of the traditional and state of the art techniques to deal with quality problems. This chapter is organized as follows: Section 1.1 introduces the concept of data quality in general and why is it so important. In section 1.3 one of the most investigated data quality problems is presented, the object identification problem. In section 1.4 some state of the art offline data cleaning methodologies are presented. Finally in section 1.5 some state of the art online data cleaning methodologies are presented.

### 1.1 An introduction to data quality

The consequences of data quality are often experienced in everyday life without making the right connection with their causes. For example, in a *Mail Service*, the wrong delivery of a letter could be attributed to a malfunction in the service while the real problem was in the misspelling of the real address or even its absence from the system database. Similarly, the duplicate delivery of an automatically generated mail is often sign of a duplicate entry in the system database.

Data quality has serious consequences for an effective and efficient functionality of organizations and businesses. The report on data quality of the Data Warehousing Institute estimates that data quality costs US businesses more than 600 billion dollars a year [Ins]. The topic is though more serious than one can think. Some examples of the importance of data quality in organizational processes are

**Customer matching** In information systems of public and private organizations, several, uncorrelated and scarcely controlled procedures are often used to update information concerning a specific customer. As a consequence such information may result inconsistent and of poor quality. For instance, it is very complex for a bank to provide clients with a unique list of their accounts and funds.

**Corporate house-holding** Many organizations establish different relations with single members of households, or more generally with related groups of people. It results very complex for them to reconstruct the household relationships in order to carry on more effective marketing strategies. This problem is even more complicated than the previous, in which the information concerned a single person at a time.

**Organizational fusion** A considerable effort is needed during fusion procedures of different organizations or units of the same organization in order to integrate their legacy information systems.

Such integration requires interoperability and compatibility at every layer of the information system.

The examples above are indicative of the growing need to integrate information across completely different data sources, an activity in which poor quality hampers integration efforts. Awareness of the importance of improving data quality is increasing in many contexts.

In the private domain the major initiatives are the following

- In 2005 IBM acquisition of Ascential Software, a leading provider of data integration tools,
- SAP has set up a project for testing in the area of DQ with the goal to build an internal methodology, with important savings in several internal business processes [Goe04],
- Oracle has enhanced its suite of products and services to support an architecture that optimizes data quality [ora], providing a framework for the systematic analysis of data aiming to increase the value of data as an important asset of any organization and to reduce the effort needed for the integration process.

In the public sector on the other hand there are two major initiatives

- In 2001 the president of the US signed into law an important new Data quality legislation, concerning “Guidelines for ensuring and maximizing the quality, objectivity, utility, and integrity of information disseminated by federal agencies”, which was named the *Data quality act*. According to this act, agencies had to report annually to the Office of Management and Budget (OMB) [omb] on the number and nature of data quality complaints received by their customers and how this complaints were handled. At the same time, a mechanism was included by which the public could petition agencies to correct information that did not meet OMB standard.
- The European directive 2003/98/CE on the reuse of public data [Par03], highlights the importance of reusing the vast data assets owned by public agencies. These agencies produce big quantities of data in many areas of activity such as social, economic, political, meteorological etc. Making available this vast quantity of data does not only help the directly connected areas but is also considered a fundamental instrument for extending the right to information as one of the basic principles of democracy. The directive suggests that data format should be independent from specific software environment and that filtering procedures must be applied periodically to guarantee high levels of reuse.

From a research perspective data quality has been addressed in different areas including statistics, management and computer science. The first to cope with problems caused by lack of quality in the data were statisticians in the late 1960’s, who proposed a mathematical theory for considering duplicates in statistical data sets. They were followed by researchers in management, who at the beginning of the 1980’s proposed solutions for detecting and eliminating duplicates in data manufacturing systems. It was as late as the beginning of 1990’s that computer scientists begin considering the problem of measuring and improving the quality of electronic data stored in databases, datawarehouses and legacy systems.

Normally, when people think about data quality they reduce the concept to accuracy, which is only one of its main aspects. For example, *Lengu*, the last name of the author of this thesis, could be misunderstood in many different ways if spelled during a telephone call: *Rengu*, *Lemgu*, *Lengyu* etc. It is important though to know that accuracy is only one of the four data quality main aspects which can be defined as positive responses to the following questions

**Accuracy** is the information correct?

**Completeness** is the information complete?

**Consistency** is the information consistent?

**Currency** is the information fresh enough?

To better understand this four aspects let's have a look at the following example. A more thorough explanation of the aspects will be given in section 1.2.

**Example 1.** *The relation in Figure 1.1 describes movies with title, director, year of production, number of remakes and the year of the last remake.*

ID	Title	Director	Year	#Remakes	LastRemakeYear
1	Casablanca	Weir	1942	3	1940
2	Dead poets society	Curtiz	1989	0	<i>null</i>
3	Rman Holiday	Wylder	1953	0	<i>null</i>
4	Sabrina	null	1964	0	1985

Figure 1.1: A *Movies* relation with data quality problems

*The relation suffers from big problems related to data quality. At first glance only the name of the third movie seems affected by misspelling errors, which is a particular type of accuracy problem. The correct title of this movie should be Roman Holidays. Another accuracy problem is related with the exchange of directors between movies 1 and 2, being Weir the director of the second and Curtiz that of the first. Other data quality problems are the missing value of the director in movie 3, which is a completeness problem, and the number of remakes of movie 4, which is a currency problem because recently a new remake of the movie was done. Finally there are two consistency problems in the database: In the first movie the year of the last remake is smaller than the year of production, which is impossible in practice, while in the fourth movie the year of the last remake cannot be different from null as the number of remakes is 0.*

The above example and previous considerations show that data quality is a multifaceted concept, composed of different aspects. It also shows that in some cases it is relatively easy to detect a data quality problem, while in others this may be almost impossible. For example it is easy to see there is a completeness problem if the name of the director is missing, but it is really complicated if a whole tuple representing a movie is missing. The above example showed the case of a single relational table with simple data type attributes. Imagine now a database or even a cluster of databases composed of a great number of tables, in which attribute values may be of more complicated types than strings or integers. It would be enormously more complicated to detect quality problems in such scenarios.

## 1.2 Data quality dimensions

In the previous section we provided an intuition of the base aspects of data quality. Let's now enter in more detail and give a more complete definition and overview of each aspects.

### 1.2.1 Accuracy

Accuracy is defined as the level of closeness between a value  $v$  and a value  $v'$ , considered as the correct representation of the real-life phenomenon that  $v$  aims to represent. For example if the name of a person is *John* then  $v' = John$  is the correct value, while  $v = Jon$  is not. There are two distinguished types of accuracy namely a syntactic accuracy and a semantic accuracy.

The *syntactic accuracy* refers to the closeness of a value  $v$  to the elements of the corresponding definition domain  $D$ . To continue with the following example, if  $v = Jack$ , even if  $v' = John$ ,  $v$  is considered syntactically correct as *Jack* is an admissible value in the names domain  $D$ . Syntactic

accuracy is defined in terms of a *comparison function* between a value  $v$  and the values in  $D$ . Edit distance is only one example of a comparison function, in which the distance between two strings  $s$  and  $s'$  is defined as the minimum number of character insertions, deletions and replacements to convert one string to the other.

Considering again the movies database in Example 1 the value *Rman Holiday* is a syntactically incorrect value, since there is no such value in the movies titles domain. The syntactically closest value to this one is *Roman Holiday* and the edit distance between the two strings is 1.

*Semantic accuracy* on the other hand is the closeness of the value  $v$  to true value  $v'$ . Considering again Example 1 the directors exchange between movies 1 and 2 is an example of semantic accuracy problem but not of a syntactic accuracy one. As a matter of fact, both *Curtiz* and *Weir* are values from the movies directors domain, but the first is not the director of *Casablanca* and the second is not the director of *Dead poets society*.

The above example shows clearly the difference between syntactic and semantic accuracy. While in the first case a comparison function could be used, in the second one a (*yes, no*) or (*correct, not correct*) function is more appropriate. Consequently, semantic accuracy coincides with the concept of *correctness*. In contrast with what happens for the syntactic accuracy, while measuring the semantic accuracy of a value  $v$ , the real value  $v'$  should be known or additional information should state that  $v$  is or is not the correct value.

From the above arguments, it comes out that semantic accuracy is typically more complex to assess than syntactic one. When it is known a priori that the rate of errors is low, and the errors result typically from typos, then syntactic accuracy tends to coincide with semantic accuracy, since typos produce values close to the true ones. As a result, semantic accuracy may be achieved by replacing an inaccurate value with the closest value in the definition domain, under the assumption that it is the true one.

## 1.2.2 Completeness

Intuitively, the completeness of a table characterizes the extent to which the table represents the corresponding world. the specific definition of completeness depends on the particular (logical) data model. We focus on the relational model, in which this data quality aspect can be characterized with respect to

1. the presence, absence and meaning of null values,
2. the validity of one of the two auto-excluding assumptions called *open world assumption* and *closed world assumption*.

In a model with null values, the presence of a null value has the general meaning of a missing value, i.e. a value that exists in the real world, but for some reason is not available. In order to characterize completeness it is important to understand why the value is missing. Indeed, a value can be missing either because it exists but is unknown, or because it does not exist at all, or because it may exist but it is not known whether the value exists or not. The following example illustrates this on a real scenario.

**Example 2.** *Figure 1.2 represents a Personal relation with attributes Name, Surname and Email. Notice that only the Email attribute of the first tuple is different from null. Let us suppose that the second person does not have an email at all. In this case no incompleteness case occurs. If the third person had an email, but this was unknown, then tuple 3 would present an incompleteness. Finally if it was not known if the fourth person had an email address or not, it would be impossible to define if in this case an incompleteness occurred.*

In logical models for databases, such as the relational model, there are two different assumptions on the completeness of data represented in a relation instance  $r$ . The *closed world assumption* (CWA)

ID	Name	Surname	Email
1	John	Smith	smith@disi.it
2	Edward	Monroe	<i>null</i>
3	Anthony	White	<i>null</i>
4	William	Clinton	<i>null</i>

Figure 1.2: A *Personal* relation with null values for the *Email* attribute

states that there are no values in the real world other than the ones represented by table  $r$ . In the *open world assumption* (OWA) we could not state neither the truth nor the falsity of the existence of values in the real world that are not actually represented in  $r$ .

There are four possible combinations emerging from the combination of the null values existence with the open-closed world assumption, from which we believe that the most interesting are the following

1. model without null values with OWA,
2. model with null values with CWA.

In the first case, in order to characterize completeness there is the need to introduce the concept of *reference relation*. Given a relation  $r$ , the reference relation of  $r$ , namely  $ref(r)$  is the relation containing all the tuples in the real world that satisfy the relational schema of  $r$ , meaning that all objects of the real world that should be represented by  $r$  are contained in  $ref(r)$ . In the absence of null values, given the cardinalities  $|r|$  of  $r$  and  $|ref(r)|$  of  $ref(r)$  the completeness of relation  $r$  could be defined as

$$C(r) = \frac{|r|}{|ref(r)|} \quad (1.1)$$

In reality it is almost always difficult to know exactly the cardinality of  $ref(r)$ .

In the second case, a model with null values with CWA, specific definitions for completeness could be provided considering the granularity of the model elements as follows

**value completeness** captures the presence of null values for some fields of a tuple,

**tuple completeness** characterizes the completeness of a tuple with respect to the values of its fields,

**attribute completeness** measures the number of null values for a specific attribute in a given relation,

**relation completeness** captures the presence of null values in the whole relation.

### 1.2.3 Currency

Another important aspect of data is how early, changes in objects of the real world are reflected in updates of the representing entities in a database. A first classification of data, depending on the frequency their value changes normally, can be the following

1. stable,
2. long-term changing,
3. frequently changing.

An example of stable data can for example be the birth dates. Once the birth date value of a person is introduced in a database, and it is known to be correct, it is never going to change again. Addresses on the other hand can be treated as long-term changing data, because in general people do not change their residence very frequently. Finally an example of frequently changing data can be the stock quotes on a particular market, such as *Dow Jones*.

*Currency* concerns how promptly data are updated. For example, in Figure 1.1, the attribute *#Remakes* of movie 4 has low currency because a remake of this movie has been made, but this information is not yet available on the database. By contrast, if the address of a person changed and immediately this change was reflected by an update in the addresses database, the data currency would be high.

A metric for the measurement of the currency value could be the following [BWPT98]

$$currency = age + (deliveryTime - inputTime) \quad (1.2)$$

where *age* measure how old data were when they were received, *deliveryTime* is the time the information product is delivered to the customer and *inputTime* is the time the data unit is obtained. As a consequence, currency is the sum of how old data were when they were received plus the time needed by the information system to make them available to the customer.

## 1.2.4 Consistency

The consistency aspect captures the violation of semantic rules defined over a set of data items, where items can be tuples of relational tables or records in a file. With reference to relational theory, *integrity constraints* are instantiations of such semantic rules.

Integrity constraints are properties that must be satisfied by all instances of a database schema. Although constraints are commonly defined on schemas, they can be checked on a specific instance of the schema that represents the actual extension of the database. There are two categories of integrity constraints

1. intrarelatational constraints,
2. interrelational constraints

Intrarelatational constraints can be related to single attributes (called *domain constraints*) or multiple attributes of the same relation. For example in a *Person* database a constraint could be imposed as *age is included between 0 and 120*. An example of a multiple attribute can be *date of death is null or smaller than date of birth*.

Interrelational constraints involve attributes of more than one relation. Consider again the *Movies* relation in Figure 1.1 and another one, called *OscarAwards*, representing the list of movies that have received Oscars together with the year the prize was assigned in the attribute *Year*. An example of interrelational constraint could be *Year of the Movies relation should be the same to that of OscarAwards*.

Most of the considered integrity constraints are *dependencies*. The main types of dependencies are the following

**Key dependency** is the simplest type of dependency. Given a relation instance  $r$  defined over a set of attributes  $A$ , we say that  $K \subseteq A$  is a *primary key* for  $r$ , if no two different tuples in  $r$  share the same values for  $K$ . If  $K$  is a primary key we say that a key dependency holds in relation  $r$ .

**Inclusion dependency** is a very common type of constraint also called a *referential constraint*. An inclusion dependency over a relation instance  $r$  states that some attributes of  $r$  are contained in other columns of  $r$  or in another relation instance  $s$ . A *foreign key* constraint is a common

type of referential constraint that states that the referring columns of one relation must be contained in the primary key columns of the referenced relation.

**Functional dependency** is a dependency between attribute values of the same tuple. Given a relational instance  $r$ , let  $X$  and  $Y$  be two nonempty sets of attributes in  $r$ .  $r$  satisfies the functional dependency  $X \rightarrow Y$ , if the following holds for every pair of tuples  $t_1$  and  $t_2$  in  $r$

$$\text{If } t_1.X = t_2.X, \text{ then } t_1.Y = t_2.Y \quad (1.3)$$

where the notation  $t_1.X$  means the projection of tuple  $t_1$  onto the attributes in  $X$ . Consider a *Person* database with attributes *name*, *surname*, *ssn* as the social security number and *child* as the name of one of his/her children. Intuitively there is a functional dependency  $ssn \rightarrow \{name, surname\}$  stating that if two tuples of the table share the same security number value, they will also share the same name and surname.

## 1.3 Object identification

In this section the *object identification* problem is described, probably the most important and the most extensively investigated data quality activity [BS06].

### 1.3.1 Overview

Let us introduce the object identification issue on an example related to an eGovernment application scenario. In this scenario different agencies manage administrative procedures related to different types of businesses on a regional bases in order to authorize specific activities and provide services, e.g. for collecting taxes. Each agency represents a set of businesses with some attributes in common and others that depend on the specific type of agency. In Figure 1.3 the same business is represented in three different agencies registries.

Agency	Identifier	Name
<i>Agency1</i>	CNCB94948DVD	Meat production of John Ngombo
<i>Agency2</i>	1034849404	John Ngombo canned meat production
<i>Agency3</i>	CNC894948DVD	Meat production in New York state of John Ngombo

Figure 1.3: Same business seen from three different agencies

Notice that the three tuples represent several differences. First of all the identifiers are different due to different policies used by the three agencies. Even in the case in which they share a common domain and meaning (see Agencies 1 and 3) some differences are present due to data entry errors. Names are also different although several common or similar parts exist. Yet the three tuples represent the same identical business!

*Object identification* is the activity during which data in the same source or in different ones is compared to decide whether they represent the same real-life entity or not.

As previously mentioned, poor data quality in a single database produces poor services quality and economic losses. Poor data quality referring to the same types of objects (e.g. people, businesses, etc) in different databases yields poor results in all applications that access the same object in different sources. This type of access is typical in many applications. For example, to discover tax frauds, different agencies can cross check their data and search for contradictions or correlations, and this can only be possible if data referring to the same object can really be identified.

Techniques dealing with the object identification problem are intrinsically connected with the data type used to represent the object. There are three main data types categories

**Simple structured data** correspond to pair of files or relational tables.

**Complex structured data** correspond to groups of logically related files of relational tables.

**Semi-structured data** such as XML marked documents.

The above example of different representations of the same business by different agencies is a simple structured data case. Imagine now, that together with the identifier and name, information about the address of the business were maintained from all the different agencies. If these information were maintained in different relational tables, that were connected to the former by referential keys than this would be a complex structured data case. Finally, if instead of relational databases, all information about business were kept in XML files, with different formats depending on the agencies internal policies, then a semi-structured data case would occur.

In relation to the above discussion two major terms are widely used in the literature: *record linkage* and *object identification*. Other used terms are *record matching* and *entity resolution* as well. *Record linkage* is generally used in the first case, on simple structured data. If record linkage is performed on a single table with the goal to detect and unify records in the relation that refer to the same real-life entity, it is called *deduplication* or *duplicate identification*. Object identification is normally used in the third case, when data is structured in XML marked documents formats.

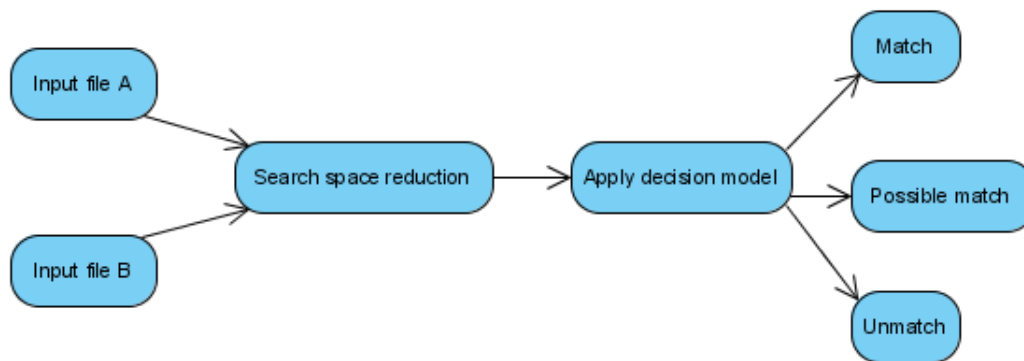


Figure 1.4: Object identification steps

Although object identification techniques strictly depends on the types of data used to represent the object, there are some common aspects shared from all of the techniques as described in Figure 1.4. For the sake of simplicity in explanation only simple structured data are considered. First data is read from both source files. After this a space reduction operation is performed starting from the Cartesian product of the input files. The reason for this is to reduce the intrinsic quadratic complexity of the method. After a subset of the Cartesian product is obtained a decision model is used to classify objects in the reduced search space as matches, i.e. corresponding to the same objects, unmatches, i.e. corresponding to different objects, and possible matches, i.e. further clerical effort is needed to manually classify the records. One of the goals of a good object identification technique is to minimize the percentage of possible matches in order to avoid unnecessary clerical involvement. Other important goals are that of minimizing false positives, i.e. number of false match decisions made, and complementary false negatives, i.e. number of false unmatch decisions made.

Finally there may be three additional phases involved in the object identification process

**A preprocessing** activity which comes before the search space reduction in the activity diagram of Figure 1.4. The goal of this activity is to make some preliminar work on data in order to eliminate inconsistencies due to different formats used and to reduce evident errors.

**The choice of a comparison function** comes immediately after the space reduction step and before the decision model application one. The goal of this activity is to choose the best comparison function to be later used by the decision model for classifying pairs of tuples.

**A verification step** during which some quality measures are performed to check if results are satisfactory. This activity is performed after several decisions have been made from the decision model. The effectiveness and efficiency of the decision model is then assessed and if one or both characteristics are considered unsatisfactory, changes are made to the model such as the type of comparison function used.

In the following subsection all but decision activities will be reported more in detail. Finally the most popular and oldest decision activity will be described in section 1.3.3.

### 1.3.2 Steps of object identification

In the following all but decision activities will be reported in detail together with some relevant examples.

#### Preprocessing

The preprocessing step includes the following activities

**conversion of upper/lower cases** In this case data corresponding to alphabetical strings are transformed to be homogeneous in terms of upper and lower cases. Consider for example business names in Figure 1.3. In some cases the first letters of any word in the agencies name are upper case while the rest is lower case, e.g. *Hewlett Packard*, while in others all letters may be in lower case, e.g. *hewlett packard*.

**replacement of null strings** Null strings must be eliminated in order to allow proper comparison. So for example *hewlett packard* must be transformed into *hewlettpackard*.

**standardization** consists in reorganization of composed fields, data type checks, replacement of alternative spellings with a single one. A simple reorganization of composed fields example could be that of an *Address* attribute. In many applications addresses are maintained as a single string, while it could be easier to compare two addresses if these were divided in several subfields such as *Street*, *CivicNumber*, *ZIPCode*, *City* and *State*. A comparison could more easily be performed on pair of values of the same subfields. Another valid reason for subfield decomposition of composed fields is the possibility of using dictionaries to perform data corrections. So for example, if the street name of an address were maintained as a separate field, a dictionary could be used to see if the name was correct and otherwise try to correct it. Data type checks regard the standardization of formats. As an example, dates must be expressed in the same format *YYYY|MM|DD*, first the year, then the month and then the day to be more easily to compare. Replacement of alternative spelling involves abbreviations that must be replaced by the corresponding complete words, e.g. *st.* by *street*.

**schema reconciliation** address the conflicts that may occur when considered data come from different data sources. Examples of such conflicts are heterogeneity conflicts, semantic conflicts, and structural conflicts.

#### Search space reduction

The object identification problem has a search space dimension equal to the cardinality of  $A \times B$ , given two sets of records  $A$  and  $B$  to be compared. The space reduction activity tries to reduce such space by one of the following techniques

**blocking** implies partitioning a set of records into mutually exclusive blocks. This means that only tuples belonging to the same block are candidates of an object identification procedure, while

those belonging to different blocks are never going to be compared. Blocking can be made either by sorting tuples on the values of a specific attribute or by hashing tuples over the value of a specific attribute. If  $n$  is the total number of tuples and  $b$  the number of blocks then each block would contain an average of  $n/b$  tuples and the total time complexity for blocking would be in  $O(h(n) + n^2/b)$ , where  $h(n)$  is the blocking procedure cost, and  $h(n) = n \log(n)$  if blocking is implemented by sorting, or  $h(n) = n$  if blocking is implemented by hashing.

**sorted neighborhood** consists of first sorting a file and then moving a window of size  $w \leq n$  over the file to compare only tuples inside the same window. Blocking reduces the search space dimension from  $n^2$  to  $wn$ . Considering the sorting complexity in  $O(n \log(n))$  the method has a time complexity of  $O(n \log(n) + wn)$ .

**pruning or filtering** has the goal of removing from the search space all objects that cannot match each other without actually comparing them. Consider the example in which two tuples  $r_i$  and  $r_j$  are considered a match if for some comparison function  $f$ ,  $f(r_i, r_j)$  is greater than a threshold  $t$ . If an upper bound for  $f$  exists, e.g.  $f(r_i, r_j) \leq g(r_i)$  for each  $r_j$ , then if  $g(r_i) \leq t$ ,  $f(r_i, r_j) \leq t$  for each  $r_j$ . If such an  $r_i$  exists it will not match with any other tuple and must be therefore eliminated from the search space.

## Comparison functions

Comparison functions have been widely investigated, especially string comparison ones [EIV07]. In the following the most important ones are described as well as examples to show similarities and differences

**edit distance** The edit distance between two strings is the minimum cost for transforming one to the other as a sequence of character insertions, deletions and replacements. A cost is assigned to each of these unitary operations in order to give the right importance to any of them. For examples supposing all unitary operations has a cost of 1, the edit distance between *Mike* and *Mikky* is 2 as the latter is obtained from the former by adding a *k* and replacing *e* by *y*.

**qgrams** First of all the substring of size  $q$  are obtained from both compared strings. The resulting sets are then compared in intersection terms, the greater the intersection size the more similar the strings. Intuitively if two strings are similar then the corresponding qgram sets will share many qgrams. 3 is a very commonly used qgram value [EIV07].

**jaccard containment and resemblance** The Jaccard coefficients measure the containment or resemblance degree between two strings (or more in general sets), using the weighted set of their qgrams or tokens. Given two weighted sets  $s_1$  and  $s_2$ , the Jaccard containment of  $s_1$  in  $s_2$  is defined to be

$$JC(s_1, s_2) = \frac{wt(s_1 \cap s_2)}{wt(s_1)} \quad (1.4)$$

while the Jaccard resemblance between  $s_1$  and  $s_2$  is defined to be

$$JR(s_1, s_2) = \frac{wt(s_1 \cap s_2)}{wt(s_1 \cup s_2)} \quad (1.5)$$

where  $wt(s)$  is the sum of weights of all elements in  $s$ .

**soundex code** The purpose of a soundex code is to cluster together words that have similar sounds. For example words *u* and *you* are pronounced in the same identical way.

**Jaro algorithm** Jaro extended the edit distance measure with another unitary operation called transposition. Intuitively this type of measure catches typos caused from transpositions of characters, a very common typing error, as a unitary operation and not as a sequence of two operations as in the edit distance case. For example names *Roald* and *Raold* have a Jaro distance of 1, but an edit distance of 2.

**Hamming distance** The Hamming distance counts the number of mismatches between two numbers. Normally it is used for numerical fixed size fields such as ZIP codes or raw binary information. For example the Hamming distance between 16128 and 16129 is 1, as the two previous Genova ZIP codes have a mismatch of a single character.

**Smith-Waterman** Given two string sequences, the Smith-Waterman algorithm uses dynamic programming to calculate the minimum cost for transforming one string to the other. Costs for single operations such as insertions, deletions and modifications are internal parameters of the algorithm. The algorithm performs very well for abbreviations, taking into account gaps of unmatched characters, and also when records have missing information or typographical mistakes.

**TF-IDF** The token frequency-inverse document frequency or cosine similarity is one of the most used comparison functions in information retrieval systems [EIV07]. The intuition is to assign higher weights to tokens appearing frequently inside the document (TF weight) and lower weights to tokens that appear frequently in the whole set of documents (IDF weight). Given a term  $i$  in a document  $j$  its weight is

$$w_{i,j} = tf_{i,j} * \log \frac{N}{df_i} \quad (1.6)$$

where  $tf_{i,j}$  is the number of occurrences of  $i$  in  $j$ ,  $df_i$  is the number of documents containing  $i$  and  $N$  is the total number of documents. The similarity between two documents is then measured as the cosine between their respective term vectors. Specifically, being  $V = \{v_1, \dots, v_n\}$  and  $U = \{u_1, \dots, u_n\}$  the weighted term vectors, the cosine similarity is

$$\frac{V * U}{|V| * |U|} \quad (1.7)$$

## Verification techniques

In the following some metrics are introduced to evaluate the effectiveness of the decision technique. A decision on actual match  $M$  or non match  $U$  on two strings can give rise to two different types of errors: *false positives*  $FP$  as records declared as  $M$  while actually being  $U$  and *false negatives*  $FN$  as records declared as  $U$  while actually being  $M$ . True positives  $TP$  are the correctly declared as  $P$  and true negatives  $TN$  the correctly declared as  $N$ . Figure 1.5 reports the several cases. From definition the following equations hold

$$M = TP + FN \quad (1.8)$$

$$U = TN + FP \quad (1.9)$$

Several indicators to evaluate the object identification techniques have been proposed, among which the most used are *recall* and *precision*. Recall measures how many true positives are identified in relation to the total number of actual matches.

$$recall = \frac{TP}{M} = \frac{TP}{TP + FN} \quad (1.10)$$

$M$	Actual match according to the real world
$U$	Actual non match according to the real world
$FP$	Match decision while actual non match
$FN$	Unmatch decision while actual match
$TP$	Match decision while actual match
$TN$	Unmatch decision while actual non match

Figure 1.5: Notations on matching decision cases

Precision on the other hand measures how many true matches are identified in relation to the total number of declared matches, including erroneous ones

$$precision = \frac{TP}{TP + FP} \quad (1.11)$$

The aim of a good decision procedure is to have both high recall and high precision values. The problem is that recall and precision are often conflicting goals in the sense that if we try to increment the number of true positives (to increase recall level) usually more false positives are also found (precision decreases) and vice versa trying to minimize the number of false positives (to increase precision level) brings a reductions of the number of true positives as a consequence (recall decreases). In order to combine recall and precision in a single indicator, the *F-measure* has been proposed which corresponds to the harmonic mean of the two former indicators

$$F\text{-measure} = \frac{2RP}{P + R} \quad (1.12)$$

Beside these specific metrics, traditional time complexity metrics are used to evaluate the efficiency of the object identification process, such as the number of comparisons performed during the process.

### 1.3.3 The Fellegi and Sunter method

The record linkage theory was first proposed by Fellegi and Sunter in [FS69]. In the following the theory is described in detail.

Given two sets of records  $A$  and  $B$ , consider the cross product  $A \times B = \{(a, b) | a \in A, b \in B\}$ . The cross product of  $A$  and  $B$  can be divided into two disjoint sets  $M$  and  $U$ , namely  $M = \{(a, b) | a \approx b, a \in A, b \in B\}$  and  $U = \{(a, b) | a! \approx b, a \in A, b \in B\}$ , where  $a \approx b$  means that records  $a$  and  $b$  represent the same real world entity, while  $a! \approx b$  means they do not. They call  $M$  the *matched set* and  $U$  the *unmatched set*. The record linkage procedure classifies all pairs of tuples as belonging to  $M$  or  $U$ . A third  $P$  set can be introduced to represent possible matches.

Suppose each record in both relations is composed of  $n$  fields. A comparison vector could be introduced to compare the field values of records  $a_i \in A$  and  $b_j \in B$ , namely  $\gamma = [\gamma_1^{ij}, \dots, \gamma_n^{ij}]$ , where  $\gamma_k^{ij} = \gamma(a_i(k), b_j(k))$  compares the  $k$ -th field of given records.  $\gamma$  is a function of the set of all  $A \times B$  record pairs, and associates to each pair a level of agreement. For example, given two relational tables with fields *name*, *surname* and *age*, we may define a comparison function on each of the fields to measure the value of agreement. For examples of comparison functions between string tuples see section 1.3.2.

Given the pair  $(a_i, b_j)$ , the following conditional probabilities can be defined

- $m(\gamma_k) = P(\gamma_k | (a_i, b_j) \in M)$ ,
- $u(\gamma_k) = P(\gamma_k | (a_i, b_j) \in U)$ .

The first value represents the probability of having a specific value for  $\gamma_k$  knowing a priori that the pair  $(a_i, b_j)$  is a match and analogously the second represents the probability of having a certain value for  $\gamma_k$  knowing that the pair  $(a_i, b_j)$  is not a valid match.

By considering all the fields, similar formulas can be defined for  $\gamma$

- $m(\gamma) = P(\gamma | (a_i, b_j) \in M)$ ,
- $u(\gamma) = P(\gamma | (a_i, b_j) \in U)$ .

The above probabilities are called *m-* and *u-probabilities* respectively. If these probabilities were known in some way then a decision procedure could be defined. Fellegi and Sunter introduced the ratio  $R$  between such probabilities as a function of  $\gamma$

$$R = \frac{m(\gamma)}{u(\gamma)} \quad (1.13)$$

The ratio  $R$  or the natural algorithm of such ratio is called a *matching weight*. By composition,  $R$  is a function of the set of all  $A \times B$  record pairs.

Fellegi and Sunter defined the following decision rule, where  $T_\mu$  and  $T_\lambda$  are two thresholds explained latter

- if  $R < T_\mu$  then the pair is considered a match,
- if  $T_\lambda \leq R \leq T_\mu$  the pair is considered a possible match,
- if  $R \leq T_\lambda$  the pair is considered a non match.

Notice that the thresholds  $T_\mu$  and  $T_\lambda$  play an essential role in the decision procedure. An important issue rises then on how to fix them. These thresholds are used to classify pairs of tuples as belonging to one of the three zones represented in Figure 1.6:  $A1$  as pairs of declared matches,  $A2$  as pairs of possible matches and  $A3$  as pairs of declared non matches. Notice that  $R$  is a ratio of probabilities. As a consequence, the assignment of a given pair  $(a_i, b_j)$  may result in a possible false assignment. *False matches* and *false non matches* are the only two types of errors allowed in the model, and  $\mu$  and  $\lambda$  represent the related error rates. By equation 1.13, high values of  $R$  correspond to low probability of false matches assignments, with such probability increasing while  $R$  decreases. Similarly, for low values of  $R$ , the probability of false non matches decreases while  $R$  decreases. The line crossing the three areas in Figure 1.6 represents a possible trend of probabilities of false matches and false non matches. In order to fix the two thresholds  $T_\mu$  and  $T_\lambda$  the accepted rates of errors for the decision procedure must be fixed first. Such error rates correspond to the two gray areas in the figure.

As a final consideration the Fellegi and Sunter theory is based on the knowledge of the *u-* and *m-* probabilities. Several methods have been proposed to estimate such probabilities. First of all, Fellegi and Sunter proposed a closed-form solution under certain assumptions. Considering that

$$P(\gamma) = P(\gamma|M)P(M) + P(\gamma|U)P(U) \quad (1.14)$$

they observed that if the comparison vector  $\gamma$  involved three fields on which a conditional independence assumption holds, then a system of seven equations and seven unknowns can be resolved to find  $P(\gamma|M)$  and  $P(\gamma|U)$ . Several other estimation methods have been proposed in the literature, the most important of which are the *expectation-minimization* algorithm [DLR77] and various machine learning methods that used labeled pairs of records to set method thresholds (*supervised learning*).

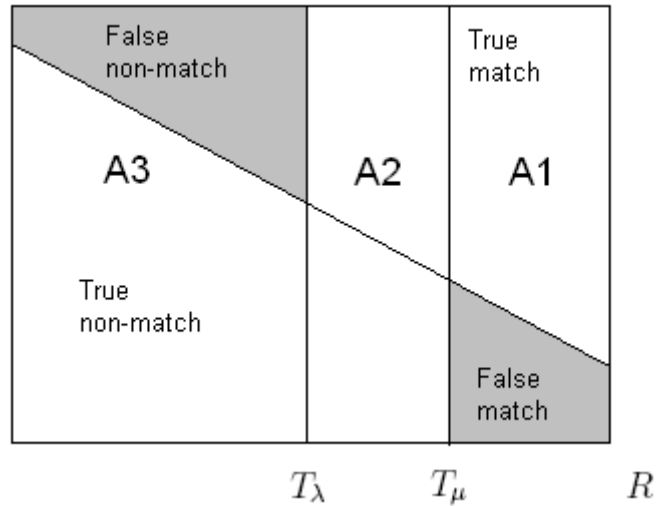


Figure 1.6: The regions of the Fellegi and Sunter model

## 1.4 Offline data cleaning

As stated in the previous section the process of object identification, one of the main approaches by which data cleaning is performed, has a high computational complexity, equal to the Cartesian product of the involved relations. In many situations a space reduction technique is not available or even in cases when it is available the complexity of data cleaning continues to be high if constraints on the technique effectiveness are imposed (e.g. *F-measure* greater than a fixed threshold  $t$ ). As a consequence, in real life scenarios it is almost impossible to perform online data cleaning. An offline approach is used instead, that performs the necessary cleaning operations before data is used for further processing. In the following three state of the art offline techniques are described in detail.

### 1.4.1 Robust and efficient fuzzy match for data cleaning

Data cleaning is the task of detecting and correcting errors on data. A prudent alternative to the expensive periodic data cleaning is to avoid the introduction of errors during the process of adding new data into the warehouse. For this reason a set of *known to be clean tuples* is used. Given a similarity function, new tuples are compared to the former ones using an error resilient matching technique called a *fuzzy match operation* [EIV03] as reported in Figure 1.7. The technique guaranties both high computational efficiency and low probability of error.

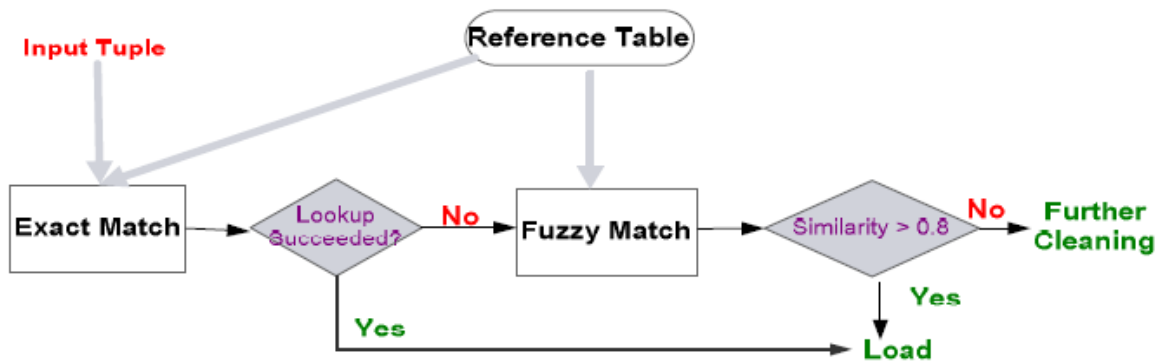


Figure 1.7: Template for using fuzzy match

Suppose an enterprise wishes to ascertain whether or not the sales record describes an existing

customer by fuzzily matching the customer attributes of the sales record against the *Customer* relation. The reference relation, *Customer*, contains tuples describing all current customers. If the fuzzy match returns a target customer tuple that is either exactly equal or *reasonably close* to the input customer tuple (as defined by a comparing function), then we would have validated or corrected, respectively, the input tuple.

The similarity between an input tuple and a reference tuple, a clean one, is the minimum cost of transforming the former into the latter, the less the cost, the higher the similarity. The allowed transforming operations are

- token replacement
- token insertion
- token deletion

where the term *token* is used in its general meaning and depending on the particular application may be a qgram of fixed [CGK06] or dynamic size [LWY07] or a single string inside a tuple such as *Corp* inside *Microsoft Corp*. Every operation has its own cost based on the type of the operation and the importance of the token in the database (IDF weights).

In order to perform the fuzzy cleaning approach on new entries, while confronting them with a set of the *clean tuples*, the *k-most similar* notion has to be introduced. Given a reference relation  $R$  and a minimum threshold  $c$ , the similarity function  $fms$  and an input tuple  $u$ , the  $k$ -most similar tuples to  $u$ , are the tuples in  $R$  having similarity with  $u$  greater than  $c$ . To efficiently solve the problem  $B+$  index trees could be used. The authors call these trees *ETI*, error tolerant index. The primary purpose of ETI is to enable, for each input tuple  $u$ , the efficient retrieval of a candidate set  $S$  of reference tuples whose similarity with  $u$  is greater than the minimum similarity threshold  $c$ . The  $k$ -most similar tuples to  $u$  are so obtained and finally the most similar tuple among these can be inserted in the warehouse as the correct value of tuple  $u$ .

As the  $fms$  operation computational complexity is relatively high, a more efficient approximation of  $fms$ , called  $fms-apx$ , is defined.  $fms-apx$  has the following properties

1. its expected value on a tuple pair  $(u, v)$  is greater than  $fms$  value on  $(u, v)$ ,
2.  $fms-apx$  upper bounds  $fms$ .

The idea is to use first  $fms-apx$  in order to reduce the search space (i.e., the number of records to compare with a given new record) and obtain the  $k$ -most similar clean records. Finally the  $fms$  operations is used on these  $k$  records only if necessary to compute the precise similarity with the new record.

There are some further extension to the algorithm

- indexing using tokens (not q-grams and so gain efficiency),
- give different weights to different columns of a tuple,
- consider the token transposition (reorder) as a new operation.

As a final considerations the algorithm provides good accuracy (up to 90%) and good efficiency, namely 2 to 3 orders of magnitude faster the *naive algorithm*, i.e. the one that compares a new tuple with every tuple in the clean set.

## 1.4.2 Efficient set joins on similarity predicates

In [SK04] an efficient, scalable and general algorithm, called *Probe-Cluster* is represented, for performing set joins on predicates involving various similarity measures like *intersect size*  $> T$ , *Jaccard-coefficient*  $> f$ , *weighted match*  $> T$ , *cosine similarity*  $> f$  and *edit distance*. Starting from a basic *inverted index based probing method* a sequence of optimizations are added that result in an improvement of one or two orders of magnitude in running time. The main contribution of the technique is to return exact answers to partial join predicates instead of approximate ones as in previous works, e.g. [EIV03]

Traditionally the most extensively used algorithms for similarity joins are

**Probe-count** from information retrieval using an inverted index [WMB99],

**Pair-count** database method similar to probe-count [BGMZ97],

**Word groups** data mining algorithm [SK04].

The Probe-count algorithm is derived from the way keyword queries are answered in Information Retrieval using an inverted index. The inverted index maps words to the list of record identifiers that contain that word. Such an index can be constructed in memory in one sequential scan of data by inserting each scanned record into the record list associated with all the words the record contains. After constructing the inverted index, the data is scanned again. For each record  $r$ , the index is probed using each word in  $r$ . This will yield a set of record lists corresponding to matching words. Each list  $l_w$  corresponding to word  $w$  of  $r$  is associated with a *weight*( $w$ ). The record lists are merged to find all records that appear in lists whose total weight is greater equal to the threshold  $T$ . Let  $n_w$  denote the number of records containing word  $w$  and  $t$  denote the average number of words per record. The Probe count algorithm requires  $O(\sum_w n_w^2 \log t)$  comparisons, since a record list  $l_w$  associated with word  $w$  is selected for merging as many times as there are records in it, that is,  $n_w$  times. Each merge operation is roughly over  $t$  lists and since a heap can be used the merge time gets multiplied by  $\log(t)$ . The memory required by Probe count to store the inverted index is  $O(\sum_w n_w)$ .

From experimental evaluation all the three algorithms result either too slow or require too much memory [SK04]. There is need for further optimization.

The first optimization is based on the idea that frequently used words, those with a high *idf*, are not so important for the algorithm. After including the optimization in the three previous algorithms, experiments show clearly that the only practical option for set overlap join is Probe-count. As a consequence, three further enhancements are proposed for the Probe-count algorithm obtaining the more efficient *Probe-cluster* algorithm

1. Single pass build and probe (as opposed to the two passes of the initial algorithm),
2. Pre-sorting data (reduce merging time step),
3. Clustering related records (group records with highly overlapping words and create index pointers to clusters instead to individual records).

A further improvement of the algorithm is the so-called *limited memory algorithm*. The inverted index used in Probe-count is proportional in size to the total word occurrences over all records. For large enough data sizes though the available memory will be incapable for holding even a compressed index. The only viable option left then is to partition the data. [SK04] shows that even in such cases the algorithm can effectively and efficiently partition data into groups of related records. A good partition strategy should minimize the number of partitions per record without making any partition very large.

The presented method avoids these two problems in two stages partitioning data on set of words as large as possible. In the first stage a compressed version of the inverted index is used (able to

fit in any available amount of memory) to create these partitions of highly related words. Then in the second stage finer grained indices are created over subsets of partitions that fit in memory and perform the joins.

Probe-cluster algorithm can be easily extended to handle not only predicates of the form *weighted set overlap*  $> T$ .

### 1.4.3 The ssjoin operator

In [CGK06] a new primitive operator which can be used as a foundation to implement similarity joins according to a variety of popular string similarity functions, and notions of similarity which go beyond textual similarity, is introduced.

The similarity join of two relations  $R$  and  $S$ , both containing a column  $A$ , is the join  $R \bowtie_{\theta} S$  where the join predicate  $\theta$  is  $f(R.A, S.A) > \alpha$ , for a given similarity function  $f$  and a threshold  $\alpha$ . Although similarity joins may be expressed in *SQL* by defining join predicates through *user-defined functions* (UDFs), the evaluation would be very inefficient as database systems usually are forced to apply UDF-based join predicates only after performing a cross product. Consequently, specialized techniques have been developed to efficiently compute similarity joins. However, these methods are all customized to particular similarity functions, e.g. [EIV03].

A general purpose data cleaning platform, which has to efficiently support similarity joins with respect to a variety of similarity functions is faced with the impractical option of implementing and maintaining efficient techniques for a number of similarity functions, or the challenging option of supporting a foundational primitive which can be used as a building block to implement a broad variety of notions of similarity. In [CGK06], *ssjoin* (*similarity set join*) is proposed as a foundational primitive and is shown to be usable for supporting similarity joins based on several string similarity functions, e.g. edit similarity, Jaccard similarity, generalized edit similarity, Hamming distance, soundex etc. Recently in [AGK06] new algorithms for *ssjoin* are proposed having two important features: they are both exact, i.e. they always produce the correct answer, and they carry precise performance guarantees.

OrgName	3-gram	Norm
Microsoft Corp	mic	12
Microsoft Corp	ier	12
Microsoft Corp	cro	12
...	...	...
Microsoft Corp	cor	12
Microsoft Corp	orp	12

**R**

OrgName	3-gram	Norm
Mcrosoft Corp	mcr	11
Mcrosoft Corp	cro	11
Mcrosoft Corp	ros	11
...	...	...
Mcrosoft Corp	cor	11
Mcrosoft Corp	orp	11

**S**

Figure 1.8: A set-overlap example

The *ssjoin* operator applies on two relations  $R$  and  $S$ , both containing columns  $A$  and  $B$ . A group of  $R.B$  values in tuples sharing the same  $R.A$  value constitutes the set corresponding to the  $R.A$  value. The *ssjoin* operator returns pairs of distinct values  $(R.A, S.A)$  if the overlap of the corresponding groups of  $R.B$  and  $S.B$  values is above a user specified threshold. Both weighted and unweighted versions are allowed. As an example, consider two relations  $R[OrgName, 3gram]$  and  $S[OrgName, 3gram]$  represented in Figure 1.8, in which together with the organization name, all *3gram* strings are also maintained. Intuitively the idea is to perform object identification in the two relations, basing on the fact that if two objects represent the real world entity, but are not exactly the same for typos or different representation policies in the two relations, their q-gram intersection

set is going to be high enough. Setting  $A = OrgName$  and  $B = 3gram$ , the `ssjoin` operator returns pairs of  $R.OrgName, S.OrgName$  values if the overlap between sets of 3grams which occur with each state is more than a given threshold. So for example the overlap between strings *Microsoft Corp* and *Mcrosoft Corp* is 10.

`ssjoin` can also be used in scenarios where different weights are assigned to different qgrams, e.g. as depending on their *idf* value to give more importance to less common qgrams. In addition a normalized threshold could be used taking in consideration the size of the two compared strings (see attribute *Norm* in Figure 1.8).

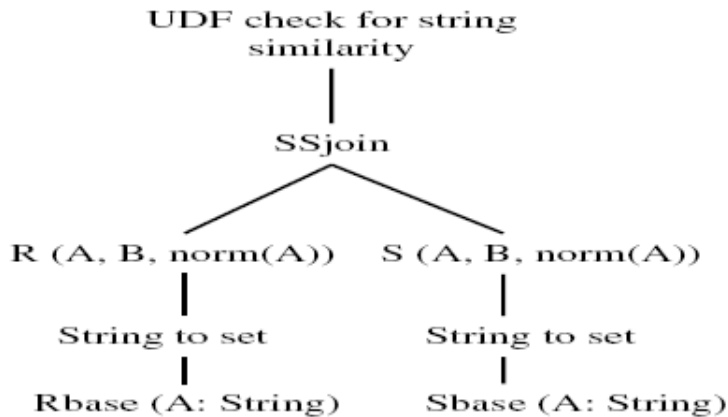


Figure 1.9: `ssjoin` as a building block operation

In Figure 1.9 the general scenario of using `ssjoin` as a base operator for defining further UDF ones is described. Without loss of generality and for the sake of clarity clarity in description,  $Rbase(A)$  and  $Sbase(A)$  unary relations are fixed, where  $A$  is a string-valued attribute. The goal is to find pairs  $(Rbase.A, Sbase.A)$  where the textual similarity defined by the given UDF is above a threshold  $\alpha$ . The approach is to first convert the strings  $Rbase(A)$  and  $Sbase(A)$  to sets, construct normalized representations  $R(A, B, norm(A))$  and  $S(A, B, norm(A))$ , and then suitably invoke the `ssjoin` operator on the normalized representations. The invocation is chosen so that all string pairs whose similarity is greater than  $\alpha$  are guaranteed to be in the result of the `ssjoin` operator. Hence, the `ssjoin` operator provides a way to efficiently produce a small superset of the correct answer. The pairs of strings are then compared using the actual similarity function, declared as a UDF within a database system, to ensure that only return pairs of strings whose similarity is above  $\alpha$  are returned.

Finally the base `ssjoin` implementation that only uses SQL-based operations such as *joins*, *group by* and *having clause* is reported in Figure 1.10. Another advantage of `ssjoin` thus results from using only SQL based operations to implement it, eliminating coding effort normally needed for integrating new operations in existing databases. First of all an equijoin operation is performed over attribute  $B$  of base relations  $R$  and  $S$ . Then a group by operation is performed over attribute  $A$  of both relations to obtain, for each relation, pairs of groups having the same value for attribute  $A$ . Finally the groups (sets) of records obtained from both relations are compared and a *having clause* is used to obtain only those with intersection size bigger than  $\theta$ .

## 1.5 Online data cleaning

All data cleaning operations defined in the previous section share a common disadvantage: A high computation complexity. As a result these methods, and many similar ones [EIV07], are performed offline, i.e. as a preprocessing step to clean new data before these are actually used by real applications

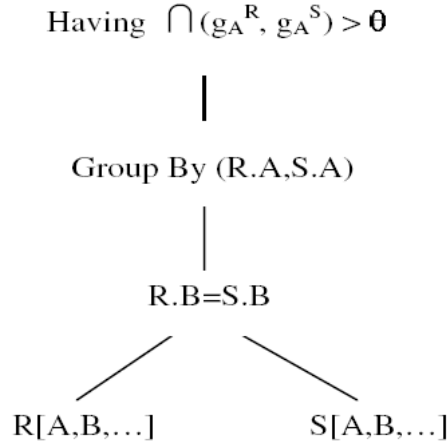


Figure 1.10: Basic ssjoin implementation

resulting in considerable gap between time data are obtained and actually processed. The rising constraint may result intolerable in many real life scenarios such as *data streaming* or *mash ups* ones, in which new data (often of poor quality) have to be immediately processed in order to be significant for applications means. In the following subsections, three state of the art online data cleaning techniques are reported as opposed to the offline ones of the previous section.

### 1.5.1 Adaptive execution of variable accuracy functions

Many analysis applications require the ability to repeatedly execute sophisticated modeling functions, which can take minutes and even hours to produce a single answer even on modern processors [Sta95]. Because of this expensive nature, such models have failed to be used in applications based on database queries, which offer a black box interface to some existing user defined functions. In [DF06] the problem of querying over sophisticated models with the developments of *VAOs* (Variable accuracy operators) is discussed. *VAOs* use a new functions interface that exposes the trade-off between compute time and accuracy existing in many modeling functions.

Previous work have been focusing on attempts to avoid such expensive function calls by either predicate reordering or caching, failing to address the problem of optimizing the execution of such functions when they still have to be run.

A trade application is an example where *VAOs* could be used. In such an example, the expensive operations to be called are those computing the real price of bonds based on models using *Partial Differential Equations* (PDEs) that can not be solved analytically and the query operations using the values returned from such functions are selection predicates or aggregates (*min*, *max*, *sum* and *average*).

Selection *VAOs* define an iterative interface between user defined functions and selection predicates. Initially some coarse bounds are returned for the result (upper and lower bound) and the value of the predicate (as a *boolean*) is computed over these bounds. If there is a final value, there is no need for further CPU cycles to compute a more precise function value. If the bounds are too coarse for the predicate to give a final result, iteration continues refining the bounds and trying to satisfy the predicate. The procedure is iterated until a final answer is returned or refining result bounds is useless for application means. This technique allows to save machine cycles, although some extra work is introduced due to communication interface and results passing.

In aggregate operators such as *min* and *max* over  $n$  objects a similar approach is used but instead of having two single bound values there are  $2n$  bounds, one for every object, and at every step *VAO* has to decide which objects bounds to refine for saving more work. For *max*, iteration continues until the greater objects lower bounds are greater than that of every other objects upper bounds or until

for application reasons there is no need to refine *interesting* objects bounds.

For *sum* and *average* operators over  $n$  weighted objects the resulting error is iterated until the total error is less than  $n * (\text{single object error})$  and at every iteration the bounds of the object are chosen as previously. In this case, gain in efficiency is proportional to how unequal the weight distribution is among the objects. If all objects have more or less the same weight, no great efficiency gain is achieved, on the other hand, if there are few objects with high weights and many with low ones, the gain in efficiency is big.

## 1.5.2 Declarative support for sensor data cleaning

Pervasive applications rely on data captured from the physical world through sensor devices, but these kinds of data tend to be unreliable, i.e. [JGF06]. Data must therefore be cleaned before applications make use of them. In [JAF<sup>+</sup>06] *Extensible sensor stream processing* (ESP) is presented, which is a framework for building sensor data cleaning infrastructures.

Dirty data from sensor networks exists in two general forms

- missed readings (low cost hardware and wireless communication employed),
- unreliable readings (fail dirty, imprecise readings, etc).

Most previous work on sensor data cleaning deals at some extent with tedious post-processing and application specific means to clean data. For example Ajax [G<sup>+</sup>01] does offline cleaning in data warehouses. The main disadvantage of these techniques is that they do not utilize the temporal or spatial aspects of data. *ESP*, instead, offers a general filtering procedure for data before being used from a particular application and uses a declarative language to be configured (ease of use and data independence). It uses the *CQL* language that can be processed by *TelegraphCQ* [C<sup>+</sup>03].

*ESP* uses *temporal* and *spatial granules* to capture time and space in sensor-based applications based on the fact that data inside a single granule is expected to be homogeneous. For example in a redwood tree temperatures scenario, sensors could be used to measure the temperature of the environment in different points. The temporal granule captures the intuition that no *enormous* fluctuations in the temperature of the same point are going to be observed in two consecutive or near points in time, while the spatial granule capture the intuition that no *enormous* fluctuations of the temperature value may occur in two near in space points.

*ESP* is designed as a simple pipeline of five stages (see Figure 1.11) each of which uses the output of the previous and produces the input for the following

**Point** single value in a sensor stream,

**Smooth** uses temporal granule to correct for missed readings and detect outliers in a single stream,

**Merge** uses application spatial granule to correct for missed readings and remove outliers spatially,

**Arbitrate** deals with conflicts such as duplicate readings,

**Virtualize** uses information from different types of sensors to improve quality.

The *ESP* effectiveness and ease of configuration is shown in three different scenarios

**RFID based scenario** inventory in a library with radio frequency identification technology,

**Environment monitoring scenario** temperature sensors in a Redwood tree,

**Digital Home scenario** motion, sound and RFID sensors to improve life quality and resources efficiency.

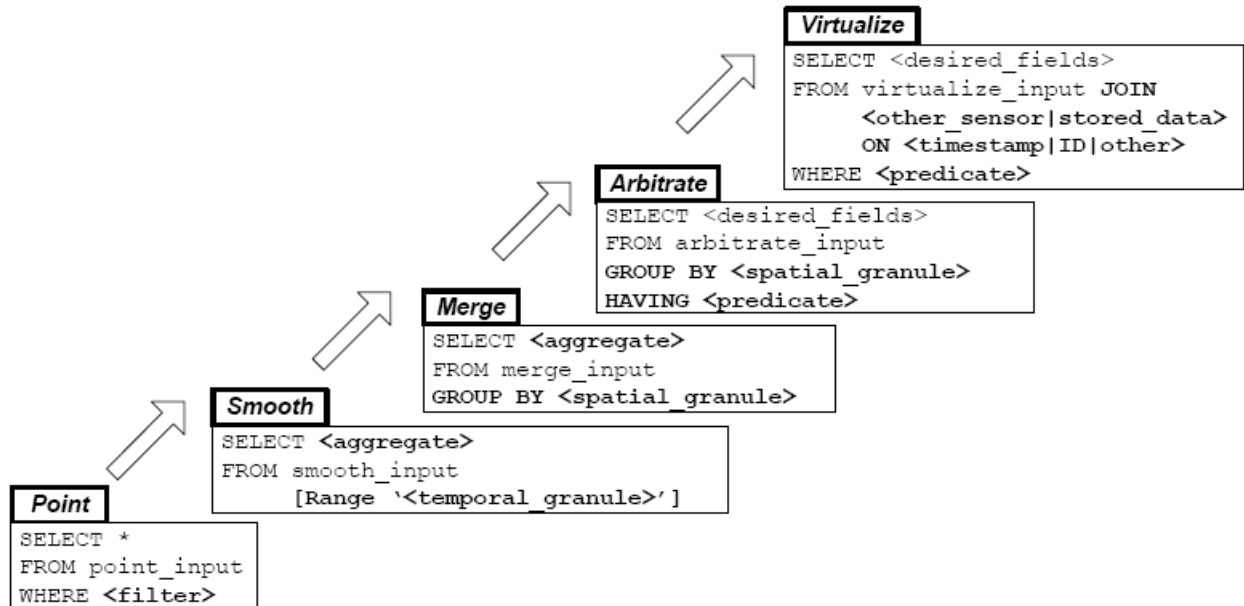


Figure 1.11: ESP processing stages

RFID (radio frequency identifications) uses low range (decimeters or a few meters) radio frequency waves to keep track of the presence or not of a determined object in a given area. Interesting objects for the applications have a radio frequency transmitter, called *tag*, while antennas, called *tag readers*, are placed in the interesting areas. The RFID technology could be used, for example, in a library to keep track of the presence of books in the shelves.

The following are some of the future possible improvements and applications of the technique

- adaptive granules for these kind of applications where a fixed size granule over time is not possible, i.e. by use of sampling theory,  $\pi$  estimators [JGF06]),
- aoft sensors, machine learning techniques to define numerical parameters,
- query-driven operations, incorporate query requirements in the infrastructure to help driving query mechanisms.

### 1.5.3 Adaptive cleaning for RFID data streams

To compensate for the inherent unreliability of RFID (radio frequency identifications) data streams, most RFID middleware systems employ a *smoothing filter*, a sliding window aggregate that interpolates for lost readings. In [JGF06] *SMURF* is introduced, the first declarative, adaptive smoothing filter for RFID data cleaning. *SMURF* models the unreliability of RFID readings by considering RFID streams as a statistical sample of the tags in the real world and exploits techniques grounded in sampling theory. Using models such as the *binomial* and  $\pi$ -estimators [S<sup>+</sup>92], *SMURF* continuously adapts the smoothing window size to provide accurate data to applications.

Previous techniques used fixed size windows decided a priori, i.e. [JAF<sup>+</sup>06]. These kind of windows fail to comply with application needs for two reasons

1. applications vary over time (there is no best fixed size window for the entire time an application runs),
2. even supposing applications were static over time, it would still be difficult to estimate the best size for the window.

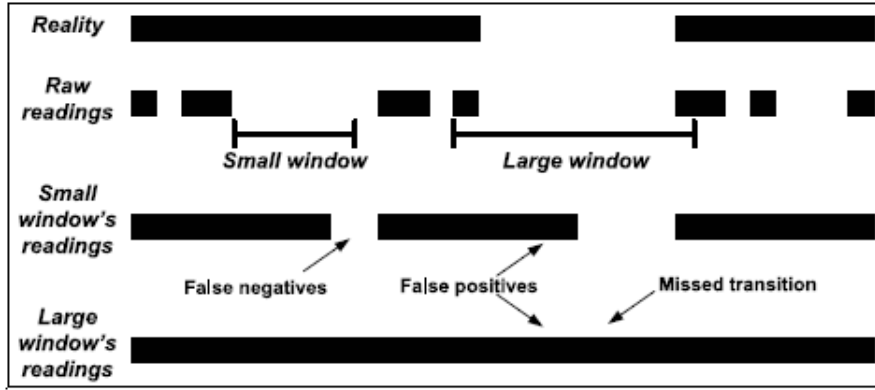


Figure 1.12: Fixed smoothing window size filters

In Figure 1.12 an RFID scenario example is given. The first horizontal black line (the one on the top) shows the theoretically perfect readings of a RFID tag, i.e. the time instants the tag was inside the radio frequency perfect range. The second line shows the actual readings made by the tag reader. Notice that many readings have been lost as the reader is not perfect. The further 2 lines show the effect of a fixed filter cleaning over the received signal

- by using a small window false negatives (i.e. missed readings) and false positives (erroneously reported readings) are introduced,
- by using a large window false positives are introduced and tag transitions are lost (tag movements out and into the reader range).

*SMURF* uses an adaptive size window in order to comply with the previous requirements and to ensure the best balance between data completeness (large window to correct reader unreliability) and tag dynamics (small window to capture tag movements).

Two novel smoothing mechanisms using *SMURF* are introduced in [JGF06]

1. Cleaning the readings of a single tag using techniques based on binomial sampling (*SMURF* employs its binomial sampling model to detect transitions as a statistically significant deviations in the observed binomial sample size from its expected value).
2. Cleaning an aggregate signal over a tag population based on  $\pi$ -estimators (transitions detected in a similar way as previously).

Experimental values show that *SMURF* succeeds to clean data giving in all cases better accuracy than filters using static fixed windows [JGF06].

# Chapter 2

## Probability and probability models

The study of probability stems from the analysis of certain games of chance and it has found applications in most branches of science and engineering [Hsu97]. In this chapter the basic concepts of probability theory are presented.

The rest of this chapter is organized as follows: in section 2.1 a brief description of probability basic concepts is given; section 2.2 introduces the notion of random variables, distribution functions and some specific distributions; in section 2.3 random processes are discussed; finally, section 2.4 describes the notion of hypothesis testing.

### 2.1 Probability basics

In this section we introduce the most important building blocks of probability such as random events and sample space.

#### 2.1.1 Sample space and events

In the study of probability, any process of observation is referred to as an *experiment*. The results of an observation are called the *outcomes* of the experiment. An experiment is called a *random experiment* if its outcome cannot be predicted. Typical examples of random experiments are the roll of a die, the toss of a coin and drawing a card from a deck.

The set of all possible outcomes of a random experiment is called the *sample space* (or *universal set*), and it is denoted by  $S$ . An element in  $S$  is called a *sample point*. Each outcome of a random experiment corresponds to a sample point.

**Example 3.** Consider the experiment of tossing a coin:

1. once
2. twice

Every time the coin is tossed there are two possible outcomes, head ( $H$ ) or tail ( $T$ ). Thus, in the first case  $S = \{H, T\}$  and the sample size cardinality is 2, in the second case  $S = \{HH, HT, TH, TT\}$  and the sample size cardinality is 4.

#### 2.1.2 Notions and axioms of probability

An assignment of real numbers to the events defined in a sample space  $S$  is known as a *probability measure*. Consider a random experiment with a sample space  $S$  and let  $A$  be a particular event defined in  $S$ ,  $A \subseteq S$ .

Suppose that the random experiment is repeated  $n$  times. If event  $A$  occurs  $n(A)$  times, then the probability of event  $A$ , denoted by  $P(A)$ , is defined as

$$P(A) = \lim_{n \rightarrow \infty} \frac{n(A)}{n} \quad (2.1)$$

where  $n(A)/n$  is called the *relative frequency* of event  $A$ . Note that this limit may be undefined and in addition, there are many situations in which the concepts of repeatability may not be valid. For any event  $A$ , the relative frequency of  $A$  will have the following properties

1.  $0 \leq n(A)/n \leq 1$ , where  $n(A)/n = 0$  if  $A$  occurs in none of the experiments and  $n(A)/n = 1$  if  $A$  occurs in all of the experiments,
2. if  $A$  and  $B$  are mutually exclusive events then  $n(A \cup B) = n(A) + n(B)$ .

Let  $S$  be a sample space and  $A, B$  events in  $S$ ,  $A, B \subseteq S$ . The axioms over which probability science is build are the following

1.  $P(A) \geq 0$
2.  $P(S) = 1$
3.  $P(A \cup B) = P(A) + P(B)$ , if  $A \cap B = \emptyset$

Based on these axioms some elementary properties are defined

1.  $P(\bar{A}) = 1 - P(A)$ , where  $\bar{A} = S \setminus A$
2.  $P(\emptyset) = 0$
3.  $P(A) \leq P(B)$ , if  $A \subseteq B$
4.  $P(A) \leq 1$
5.  $P(A \cup B) = P(A) + P(B) - P(A \cap B)$

A sample space is said to be made of equally likely events if all events in the sample space have the same probability of occurrence. This concludes our brief introduction to the concepts and properties of probability.

## 2.2 Random variables

In this section we introduce the notion of random variable and some of its important properties such as the distribution function. At the end of the section the characteristics of some particular discrete random variables are described in detail. We will make use of these specific variables in the remainder of the thesis to explain the functionality of parts of our system.

### 2.2.1 Definitions

The purpose of random variables is to introduce some probabilistic functions that make it easier to compute the probabilities of various events.

Consider a random experiment with sample space  $S$ . A *random variable* that describes the experiment is a function  $X : S \rightarrow \mathcal{R}$ , that assigns a *real* number to any event  $\zeta \subseteq S$ . It is clear that a random variable is not a real variable but a function, the terminology has been forced by tradition [Hsu97]. The sample space  $S$  is named the *domain* of the random variable  $X$ , and the collection of all numbers from the *codomain* is named the *range* of the random variable  $X$ . Thus, the range of  $X$  is a certain subset of the set of all real numbers (Fig. 2.1).

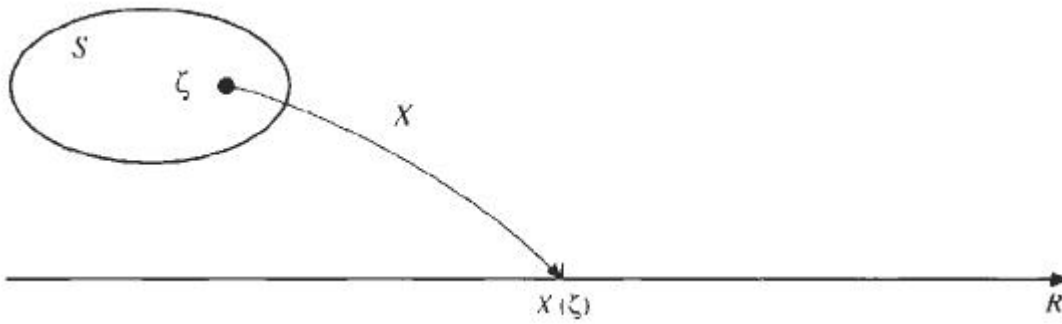


Figure 2.1: A random variable  $X$

**Example 4.** Consider the experiment of tossing a coin once. The sample space of the experiment is  $S = \{H, T\}$ . The random variable  $X$  could be defined as  $X(H) = 0$  and  $X(T) = 1$ .

If  $X$  is a random variable and  $x, x_1, x_2$  are real numbers, then the following events can be defined

1.  $(X = x) = \{\zeta \in S : X(\zeta) = x\}$
2.  $(X \leq x) = \{\zeta \in S : X(\zeta) \leq x\}$
3.  $(X > x) = \{\zeta \in S : X(\zeta) > x\}$
4.  $(x_1 < X \leq x_2) = \{\zeta \in S : x_1 < X(\zeta) \leq x_2\}$

**Example 5.** Consider the experiment of tossing a fair coin 3 times. Then the sample space of this experiment will consist of eight equally like sample points  $S = \{HHH, \dots, TTT\}$ <sup>1</sup>. If  $X$  is the random variable that gives the number of heads obtained

1.  $P(X = 2) = 3/8$ . Let  $A$  be the event defined by  $X = 2$ , then  $A = \{THH, HTH, HHT\}$ .
2.  $P(X < 2) = 1/2$ . Let  $B$  be the event defined by  $X < 2$ , then  $B = \{TTT, HTT, THT, TTH\}$ .

## 2.2.2 Distribution functions

The *distribution functions* or *cumulative distribution function* of a random variable  $X$  is defined as

$$F_X(x) = P(X \leq x) \quad -\infty < x < \infty \quad (2.2)$$

Most of the information about the random experiment described from a variable  $X$  can be obtained from its distribution function. The distribution function has the following properties

1.  $0 \leq F_X(x) \leq 1$
2.  $F_X(x_1) \leq F_X(x_2)$ , if  $x_1 \leq x_2$
3.  $\lim_{x \rightarrow \infty} F_X(x) = F_X(\infty) = 1$
4.  $\lim_{x \rightarrow -\infty} F_X(x) = F_X(-\infty) = 0$
5.  $\lim_{x \rightarrow a^+} F_X(x) = F_X(a^+) = F_X(a)$

Property 1 follows from the fact that  $F_X(x)$  is a probability, property 2 states that  $F_X(x)$  is a monotonically increasing function. Property 5 states that  $F_X(x)$  is *continuous on the right*.

<sup>1</sup>All the outcomes have the same probability of occurrence because the coin is fair.

**Example 6.** Consider again the random variable defined in Example 5. Table 2.1 gives the values of  $F_X(x)$  for integer values in the range  $-1 \dots 4$ . As the values of  $X$  can only be integers the distribution function assigns to non-integer values the value of the first smaller integer. Figure 2.2 shows the graph of  $F_X(x)$ .

Table 2.1: Distribution function values

$x$	$X \leq x$	$F_X(x)$	$p_X(x)$
-1	$\emptyset$	0	0
0	$\{TTT\}$	$1/8$	$1/8$
1	$\{TTT, HTT, THT, TTH\}$	$1/2$	$3/8$
2	$\{TTT, HTT, THT, TTH, HHT, HTH, THH\}$	$7/8$	$3/8$
3	$S$	1	$1/8$
4	$S$	1	0

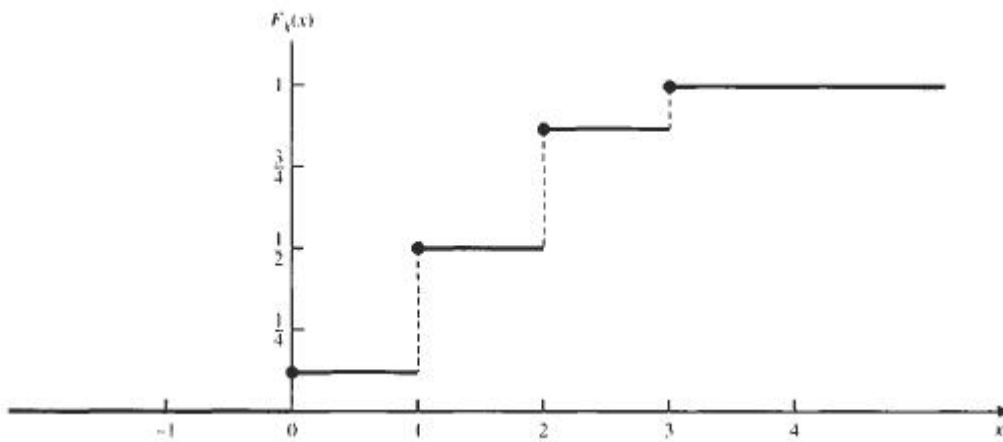


Figure 2.2: Distribution function graph

The important point here is that given the random variable  $X$ , for which  $F_X$  is known, the following probabilities and others can be computed

1.  $P(a < X \leq b) = F_X(b) - F_X(a)$
2.  $P(X > a) = 1 - F_X(a)$
3.  $P(X < b) = F_X(b^-)$

$F_X(b^-)$  is a compact way to denote  $P(X < b)$  which is equal to  $F_X(b) - P_X(b)$  if  $F_X(b)$  is defined.

### 2.2.3 Discrete random variables

A variable  $X$  is called a *discrete random variable* if the range of  $X$  is made either of finite or of infinite numerable points. The cumulative distribution function graph of a random variable has the particular scale-like look, meaning that in a given interval the graph is either constant or makes a discontinuous jump. The variable in Example 5 is a discrete random variable.

The *probability mass function* of a random variable  $X$  is defined as

$$p_X(x) = P(X = x) \tag{2.3}$$

This function assigns a probability value to each point  $x_1, x_2 \dots$  in the variable range.

$p_X(x)$  has the following properties

1.  $0 \leq p_X(x_k) \leq 1$ , for  $x = x_1, x_2, \dots$
2.  $p_X(x) = 0$ , if  $x \neq x_1, x_2, \dots$
3.  $\sum_k p_X(x_k) = 1$

The cumulative distribution function of a discrete random variable can be obtained by

$$F_X(x) = \sum_{x \leq x_k} p_X(x_k) \quad (2.4)$$

### 2.2.4 Mean and variance

The  $n^{\text{th}}$  moment of the discrete random variable  $X$  is defined by

$$E(X^n) = \sum_k x_k^n p_X(x_k) \quad (2.5)$$

The mean of a discrete random variable  $X$ , denoted by  $\mu_X$  is its first moment

$$\mu_X = E(X) = \sum_k x_k p_X(x_k) \quad (2.6)$$

The intuitive meaning of  $\mu_X$  is the following: Suppose that the domain of the discrete random variable  $X$  assumes  $x_1, x_2, \dots$  possible values with probability  $p_X(x_1), p_X(x_2), \dots$ . Then the more we repeat the random experiment the more the average of the observed values gets closer to the mean. This is also known as the principle of *The law of large numbers* [Hsu97].

The second moment of a discrete random variable is the *variance*, which is defined as

$$\sigma_X^2 = \text{Var}(X) = E[X - E(X)]^2 \quad (2.7)$$

Thus,

$$\sigma_X^2 = \sum_k (x_k - \mu_X)^2 p_X(x_k) \quad (2.8)$$

The variance of a random variable is a measure of *statistical dispersion*, averaging the square distance of all possible values from the expected value.

The *standard deviation*  $\sigma_x$  is the square root of the variance.

**Example 7.** Consider again the random variable  $X$  defined in Example 5 which counts the number of heads obtained if the coin is tossed three times. The probabilities of each possible outcome from 0 to 3 are given in Example 6. The mean of this random variable is

$$\begin{aligned} \mu_X &= \sum_k x_k p_X(x_k) \\ \mu_X &= 1/8 * 0 + 3/8 * 1 + 3/8 * 2 + 1/8 * 3 = 12/8 = 3/2 \\ \mu_X &= 1.5 \end{aligned}$$

The variance of this random variable is

$$\begin{aligned} \sigma_X^2 &= \sum_k (x_k - \mu_X)^2 p_X(x_k) \\ \sigma_X^2 &= (0 - 1.5)^2 * 1/8 + (1 - 1.5)^2 * 3/8 + (2 - 1.5)^2 * 3/8 + (3 - 1.5)^2 * 1/8 \\ \sigma_X^2 &= 9/4 * 1/8 + 1/4 * 3/8 + 1/4 * 3/8 + 9/4 * 1/8 = 24/32 = 3/4 \\ \sigma_X^2 &= 0.75 \end{aligned}$$

## 2.2.5 Entropy of a discrete finite probability distribution

A length  $N$  probability distribution is a vector  $P = (p_1, p_2, \dots, p_N)$ , in which every  $0 \leq p_i \leq 1$  and  $\sum_{i=1}^N p_i = 1$ . Remember from the previous subsections, that a length  $N$  random variable is a vector  $X = (x_1, x_2, \dots, x_N)$  in which every component is a real number,  $\forall i. x_i \in \mathcal{R}$ . For a fixed length  $N$  random variable  $X$  and a length  $N$  probability distribution  $P$ , the numbers  $x_i$  are interpreted as possible values of  $X$ , and the quantities  $p_i$  as the probability of occurrence of the event  $A = \{X = x_i\}$ .

Remember also, from the previous subsections, that the average value of a random variable  $X$ , with respect to a probability distribution  $P$  is given by  $\mu_X = \sum_{i=1}^N x_i p_i$ .

In particular, to any probability distribution  $P = (p_1, p_2, \dots, p_N)$ , it is possible to associate the information random variable  $I(P) = (I(p_1), I(p_2), \dots, I(p_N))$ , where  $\forall p \in P. I(p) = -\log p$ , whose mean with respect to the probability distribution  $P$ , namely the *entropy of the probability distribution*, denoted by  $H(P)$ , is expressed by the following formula (with the convention  $0 \log 0 = 0$ )

$$H(P) = - \sum_{i=1}^N p_i \log p_i \quad (2.9)$$

with  $0 \leq H(P) \leq \log N$ . The real number  $I(p) = -\log p$  is a measure of uncertainty due to the knowledge of a probability, since if the probability is 1, then there is no uncertainty and its corresponding measure is 0.

## 2.2.6 Some special distributions

In this subsection we introduce some important distributions such as the Bernoulli, binomial and the hypergeometric ones. We will use these distributions in Chapter 5 to explain parts of our system.

### The Bernoulli distribution

A random variable  $X$  is called a *Bernoulli random variable with parameter  $p$*  if its pmf is given by

$$p_X(k) = p^k (1-p)^{1-k} \quad k = 0, 1 \quad (2.10)$$

where  $0 \leq p \leq 1$ . The cumulative distribution function of the Bernoulli random variable  $X$  is given by

$$F_X(x) = \begin{cases} 0 & x < 0 \\ 1-p & 0 \leq x < 1 \\ 1 & x \geq 1 \end{cases} \quad (2.11)$$

The mean and the variance of the Bernoulli random variable are

$$\mu_X = p \quad (2.12)$$

$$\sigma_X^2 = p(1-p) \quad (2.13)$$

The Bernoulli distribution is used to describe the class of experiments in which the outcome is classified either as a success with probability  $p$  or as a failure with probability  $1-p$ .

**Example 8.** *The experiment of tossing a coin can be thought as a Bernoulli experiment in which obtaining a Head (Tail) is considered a success and obtaining a Tail (Head) is considered a failure. In this case  $p = 1/2$ .*

## The binomial distribution

A random variable  $X$  is called a *binomial random variable with parameters  $n$  and  $p$* ,  $B(n, p)$ , if its probability mass function is given by

$$p_X(k) = \binom{n}{k} p^k (1-p)^{n-k} \quad k = 0, 1, \dots, n \quad (2.14)$$

where  $0 \leq p \leq 1$  and

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

is the binomial coefficient. The corresponding cumulative distribution function of  $X$  is

$$F_X(x) = \sum_{k=0}^x \binom{x}{k} p^k (1-p)^{x-k} \quad x = 0, \dots, n \quad (2.15)$$

The mean and the variance of the binomial random variable are

$$\mu_X = np \quad (2.16)$$

$$\sigma_X^2 = np(1-p) \quad (2.17)$$

The binomial distribution is used to describe experiments in which  $n$  independent Bernoulli trials are performed in sequence. The random variable in this case measures the number of successes in these trials.

**Example 9.** *If we think about the experiment of tossing a coin 10 times, then a random variable  $X$  that measures the number of Heads (Tails) in these trials is a binomial random variable with  $p = 1/2$  and  $n = 10$*

## The Hypergeometric distribution

A random variable  $X$  is called a *hypergeometric random variable with parameters  $N$ ,  $k$  and  $n$* ,  $H(N, k, n)$ , if its probability mass function is given by

$$p_X(x) = \frac{\binom{k}{x} \binom{N-k}{n-x}}{\binom{N}{n}} \quad (2.18)$$

Unfortunately there is not a compact formula to calculate the cumulative distribution function of the hypergeometric distribution, it can only be computed using equation 2.2, which can be computationally expensive for large values of the hypergeometric parameters.

The mean and the variance of the hypergeometric random variable are

$$\mu_X = \frac{nk}{N} \quad (2.19)$$

$$\sigma_X^2 = \frac{n \frac{k}{N} \frac{N-k}{N} (N-n)}{N-1} \quad (2.20)$$

The hypergeometric distribution is used to describe those experiments in which  $n$  draws are made from a population of size  $N$  without replacement, in which  $k$  objects are *defective*. This

means that the objects in the population are classified in two big categories: defectived and not defectives. The random variable measures the number of defectives present in the drawn sample. To see the connection between this distribution and the binomial one, we can think of the second one as describing a sequence of  $n$  draws from a sample of size  $N$  with replacement, in which  $k$  objects are defective. In this case, the binomial distribution would have parameter  $p = \frac{k}{N}$ , mean  $\mu_X = \frac{nk}{M}$  that is the same of the hypergeometric and variance  $\sigma_X^2 = \frac{nk(N-k)}{N^2}$ , if we consider the event of drawing a defective object as a success.

**Example 10.** *Think about the experiment of drawing without replacement 10 balls from a bag of 50, 30 of which are black. The random variable that counts the number of black balls in the drawn sample is a hypergeometric random variable with parameters  $N = 50$ ,  $k = 30$  and  $n = 10$ .*

## 2.3 Random processes

A random process is the mathematical model of an empirical process whose development is governed by probability laws. Understanding such models provides powerful tools to analyse and control many engineering complex processes.

A *random process* is a family of random variables  $\{X(t), t \in T\}$  defined on a given probability space, indexed by the parameter  $t$ , where  $t$  varies over an *index set*  $T$ . Recall that a random variable is a function defined on the sample space  $S$ . Thus, a random process  $\{X(t), t \in T\}$  is actually a function of two arguments  $\{X(t, \zeta) : t \in T, \zeta \in S\}$ . For a fixed  $t = t_k$ ,  $X(t_k, \zeta) = X_k(\zeta)$  is a random variable denoted by  $X(t_k)$ , as  $\zeta$  varies over the sample space  $S$ . On the other hand, for a fixed sample point  $\zeta_i \in S$ ,  $X(t, \zeta_i) = X_i(t)$  is a single function of time  $t$ , called a *sample function* or a *realization of the process*. The totality of all sample functions is called an *ensemble*. Of course if both  $\zeta$  and  $t$  are fixed,  $X(t_k, \zeta_i)$  is simply a real number.

In a random process  $\{X(t), t \in T\}$ , the index set  $T$  is called the *parameter set* of the random process. The values assumed by  $X(t)$  are called *states*, and the set of all possible values forms the *state space*  $E$  of the random process. If the index set  $T$  of a random process is discrete, then the process is called a *discrete-parameter* (or *discrete-time*) process. A discrete-parameter process is also called a *random sequence* and is denoted by  $\{X_n : n = 1, 2, \dots\}$ .

Consider a random process  $X(t)$ . For a fixed time  $t_1$ ,  $X(t_1) = X_1$ , is a random variable, and its cumulative distribution function  $F_X(x_1; t_1)$  is defined as

$$F_X(x_1; t_1) = P\{X(t_1) \leq x_1\} \quad (2.21)$$

$F_X(x_1; t_1)$  is known as the *first-order distribution* of  $X(t)$ . In general the  $n^{\text{th}}$  *order distribution* of  $X(t)$  is defined by

$$F_X(x_1, \dots, x_n; t_1, \dots, t_n) = P\{X(t_1) \leq x_1, \dots, X(t_n) \leq x_n\} \quad (2.22)$$

If  $X(t)$  is a discrete time process, then it is specified by a collection of probability mass functions

$$F_X(x_1, \dots, x_n; t_1, \dots, t_n) = P\{X(t_1) = x_1, \dots, X(t_n) = x_n\} \quad (2.23)$$

**Example 11.** *Suppose there is a population of size  $N$  in which  $k$  elements are defective. Suppose  $n$  draws are made from the population without replacement. Let  $T = t_1, \dots, t_n$  be the instants of time in which every draw occurs in order. The random process  $\{X(t) : t = 0, \dots, n\}$  is a hypergeometric process if, for each  $t$ ,  $X(t)$  counts the number of defectives drawn to moment  $t$ .*

## 2.4 Hypothesis testing

There are many situations in which we have to make decisions based on observations of data that are random variables. The theory behind the solutions to these situations is known as *decision theory* or *hypothesis testing*.

A *statistical hypothesis* is an assumption about the probability law of a random variable. Suppose we observe a random sample  $(X_1, \dots, X_n)$  of a random variable  $X$  whose cumulative distribution function  $f(x; \theta) = f(x_1, \dots, x_n; \theta)$  depends on a parameter  $\theta$ . We wish to test the assumption  $\theta = \theta_0$ , against the assumption  $\theta = \theta_1$ . The assumption  $\theta = \theta_0$ , is denoted by  $H_0$  and is called the *null hypothesis*. The assumption  $\theta = \theta_1$  is denoted by  $H_1$  and is called the *alternative hypothesis*.

A hypothesis is called *simple* if all parameters are specified exactly. Otherwise it is called *composite*. Thus, suppose  $H_0$  corresponds to  $\theta = \theta_0$  and  $H_1$  corresponds to  $\theta \neq \theta_0$ ; then  $H_0$  is simple and  $H_1$  is composite.

Hypothesis testing is a decision process establishing the validity of a hypothesis. We can think of the decision process as dividing the observation space  $\mathcal{R}^n$  (*Euclidean  $n$ -space*) into two regions  $R_0$  and  $R_1$ . Let  $x = (x_1, \dots, x_n)$  be the observed vector. Then if  $x \in R_0$ , we will decide on  $H_0$ ; if  $x \in R_1$  we decide on  $H_1$ . The region  $R_0$  is known as the *acceptance region* and the region  $R_1$  as the *rejection* (or *critical*) *region* (since the null hypothesis is rejected). Thus, with the observation vector (or data), one of the following four actions can happen

1.  $H_0$  true; accept  $H_0$
2.  $H_0$  true; reject  $H_0$  (or accept  $H_1$ )
3.  $H_1$  true; accept  $H_1$
4.  $H_1$  true; reject  $H_1$  (or accept  $H_0$ )

The first and third actions correspond to correct decisions, while the second and fourth actions correspond to errors. The errors are classified as:

1. Type I error: Reject  $H_0$  when  $H_0$  holds
2. Type II error: Reject  $H_1$  when  $H_1$  holds

Let  $P_I$  and  $P_{II}$  denote the probabilities that error I and II happen, respectively

$$P_I = P(x \in R_1; H_0) \tag{2.24}$$

$$P_{II} = P(x \in R_0; H_1) \tag{2.25}$$

$P_I$  is denoted by  $\alpha$  and is known as the *level of significance*,  $P_{II}$  is denoted by  $\beta$  and is known as the *power of the test*. Note that since  $\alpha$  and  $\beta$  represent probabilities of events from the same decision problem, they are not independent of each other or of the sample size  $n$ . It would be desirable to have a decision process such that both  $\alpha$  and  $\beta$  will be small. However, in general, a decrease in one type of error leads to an increase in the other type for a fixed sample size. The only way to simultaneously reduce both types of error is to increase the sample size [Hsu97]. One might also attach some relative importance (or cost) to the four possible courses of action and minimize the total cost of the decision.

**Example 12.** Suppose a manufacturer of memory chips observes that the probability of chip failure is  $p = 0.05$ . A new procedure is introduced to improve the design of chips. To test this new procedure, 200 chips could be produced using this new procedure and tested. Let random variable  $X$  denote the number of these 200 chips that fail. We set the test rule that we would accept the new procedure if  $X \leq 5$ . Let

1.  $H_0 : p = 0.05$  (No change hypothesis)

2.  $H_1 : p < 0.05$  (Improvement hypothesis)

If we assume that these tests using the new procedure are independent and have the same probability of failure on each test, then  $X$  is a binomial random variable with parameters  $(n, p) = (200, p)$ .

We make a Type I error if  $X \leq 5$  when in fact  $p = 0.05$ . Thus, using equation 2.15, we have

$$P_I = P(X \leq 5; p = 0.05) = \sum_{k=0}^5 \binom{200}{k} 0.05^k 0.95^{200-k}$$

$$P_I \approx 0.067$$

Note that  $H_0$  is a simple hypothesis but  $H_1$  is a composite hypothesis.

# Chapter 3

## Symmetric set hash join algorithm

In this chapter we introduce symmetric set hash join (*sshjoin*), an approximate join algorithm suitable for stream join scenarios, in which the information about the streams is incomplete until the execution end. This type of algorithm should be incremental and begin returning results even before streams have completely been consumed. As we will see the algorithm is suitable for pipelined architectures and follows the model described in section 3.1. This chapter is organized as follows: Section 3.1 introduces a pipelined architecture for the evaluation of incremental query operators. In section 3.2 the data structures used by the symmetric set hash join algorithm are presented. Section 3.3 describes the algorithm in detail. Section 3.4 contains an analysis of the computational and spatial cost of the algorithm. Finally, section 3.5 reports the advantaged and disadvantages of the algorithm.

### 3.1 Pipelined operators

In relational algebra a query is specified as an expression in which some logical operators are applied to the base relations. Every operator takes as input one or more relations and returns as result a new relation. The relational operators are logical operators, meaning that their behavior is perfectly defined, but that there is need to implement these operators in real database systems. A physical operator is an algorithm, together with all data structures used, that performs the operation defined by the logical operator. There can be more than one physical operators implementing a single logical one, e.g. *merge join* and *hash join* both implement the *logical join* operator. The important thing is that logically thee physical operators are identical, i.e. the result produced is the same for all possible inputs (relations).

The physical counter-part of a relational expression is though a tree of physical operators, having as leaves the relational tables and in which every intermediate node corresponds to an operator, its children to the input operators and its parent to the output operator. The root of the tree is going to be the new relation defined by the given expression (see Figure 3.1 for the expression  $(R \bowtie S) \bowtie (T \bowtie U)$ ).

In traditional architectures evaluation is performed bottom-up: an operator begins returning result tuples only after its input operators have entirely finished calculating their respective results. In Figure 3.1 this means that result tuples can only be seen after the join between tables  $R$  and  $S$  and that between  $T$  and  $V$  have been entirely executed. In pipelined architectures, on the other hand, an operator reads single tuples from its input operators and returns single tuples as its own results. This type of operators are called *incremental*. In Figure 3.1 this means that the root operator could begin computing, and potentially returning, results, from the moment that each of  $R \bowtie S$  and  $T \bowtie U$  joins have returned a single result tuple.

In this thesis, pipelined evaluation is assumed to be implemented using the *iterator* model [Gra93], which has three principal functions: *open*, *next* and *close*. The *open* function prepares the operator

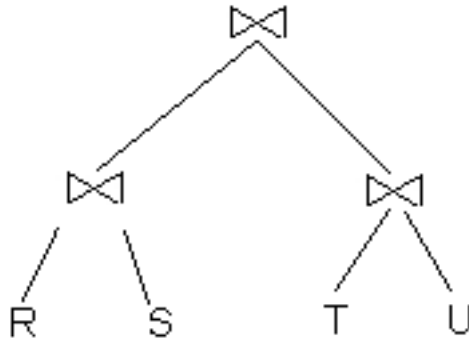


Figure 3.1: An algebra expression tree

for result production. The *next* function produces the results one at a time, and the *close* function performs cleaning up.

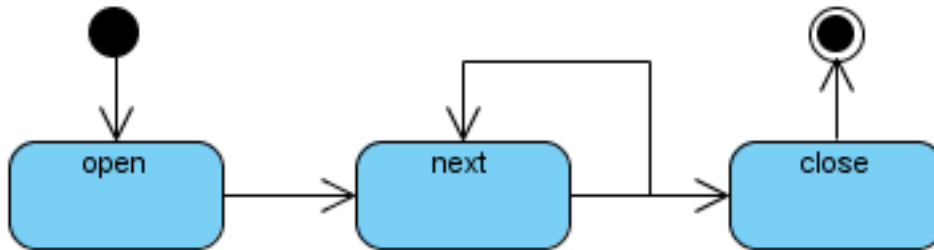


Figure 3.2: The iterator model

A state-transition diagram can be used to capture the evaluation trace of operators implemented using the iterator model (Figure 3.2). The lifecycle of a physical operator is composed of the following steps:

1. call *open* method to allocate the necessary data structures and prepare for execution,
2. iteratively call *next* method returning a single result tuple for each single call,
3. if all result tuples have been returned call *close* method to deallocate data structures used.

In Kwanchai’s architecture [EFP06] the methods set is extended by another one, called *isQuiescent*. This method returns a boolean value over the internal state of the operator. A *true* value means that the physical operator is in a safe state, its execution could be suspended and the operator replaced with another operator, that is physically different but logically identical to the previous one. After the replacement, execution could continue from the interruption point without having to recompute results, but only concentrating on the work still to be done. The possibility of changing the physical operator in the middle of a query execution without having to recompute results and still guaranteeing the soundness of the final result is very important in adaptive query processing. The most appropriate physical operators could though be chosen at every execution point in order to maximize certain parameters (e.g. ,total execution time).

## 3.2 Overview and data structures used

We are now ready to introduce the symmetric set hash join (*sshjoin*), an approximate join algorithm suitable for stream join scenarios, in which the information about the streams is incomplete until the execution end. We chose a representant from all the approximate join algorithms and another one from the stream based ones. The *sshjoin* was then retrieved melting the best characteristics of both algorithms.

Among the approximate join algorithms proposed in the literature, we chose the similarity set join (*ssjoin*) algorithm [CGK06] because of the following characteristics

- it is easy to implement,
- it serves as a building block for a large series of approximate join algorithms [CGK06].

The second quality of the algorithm is also the most interesting. Once the *ssjoin* algorithm has been implemented, if necessary this could be used to further implement other approximate join algorithms with minimal effort. The similarity set join algorithm compares two strings relying on the similarity between their *qgram sets*. A *qgram* of size  $n$  of a string  $s$  is any substring of size  $n$  of  $s$ . The *qgram set* of size 2 of the string *Roald* is  $\{Ro, oa, al, ld\}$ . According to the similarity set join philosophy, if two strings represent the same real life entity they will also share many *qgrams*.

The symmetric hash join (*shjoin*) algorithm on the other hand, is the most used for streams. As introduced in section [WA91] this algorithm has a similar behavior to the normal hash join one with some peculiarities. First of all the normal join uses only one hash table to calculate its results, while the symmetric hash join uses two tables, one for each input. Second, the normal join is a blocking operator and as a consequence not suitable for pipelined architectures. To begin calculating results it has to entirely build a hash table over one of the inputs, meaning that no result is returned before at least one of the tables has been completely read. The symmetric hash join on the other hand is a non blocking operator. The two hash tables are built in parallel while reading the tuples from both inputs. Results could be returned at any point of the join execution, using the partial information contained in the tables, without having to wait for the input total exhaustion. *shjoin* supports the traditional iterator-based, pipelined *next* operation, as follows

- If there is an outstanding *probe* tuple for which not all matches have been returned yet (which means, according to the terminology we adopt from [EFP06], that we are not in a quiescent state), the next match for that probe tuple is returned.
- If there are no outstanding tuples (i.e., if we are in a quiescent state), then a new input tuple is obtained, inserted into the appropriate hash table, and probed against the other hash table.

The main difference between the old symmetric hash join algorithm and the new symmetric set hash is the hash table structure. The *ssjoin* uses a hash table based on attributes, meaning that the hash function is evaluated only once per tuple, over the value of the join attribute, and the tuple is stored in the corresponding hash entry once. In the *sshjoin*, instead, the hash function is evaluated several times for the same tuple, exactly once for every *qgram* contained in the tuple join attribute, and the tuple is stored the same number of times in the corresponding *qgram* hash table. In Figure 3.3 the structure of this hash table is reported. In the figure, for every tuple in the *qgram* hash table additional information is maintained in the *counter* field. As we will see in the following of this section the field value is used to compute the similarity between the join attributes of two tuples.

## 3.3 Algorithm

In Figure 3.4 a simplified version of our *sshjoin* operator *next* method is reported. As we can see, the method takes 2 input parameters, resulting in a different interface from the one previously

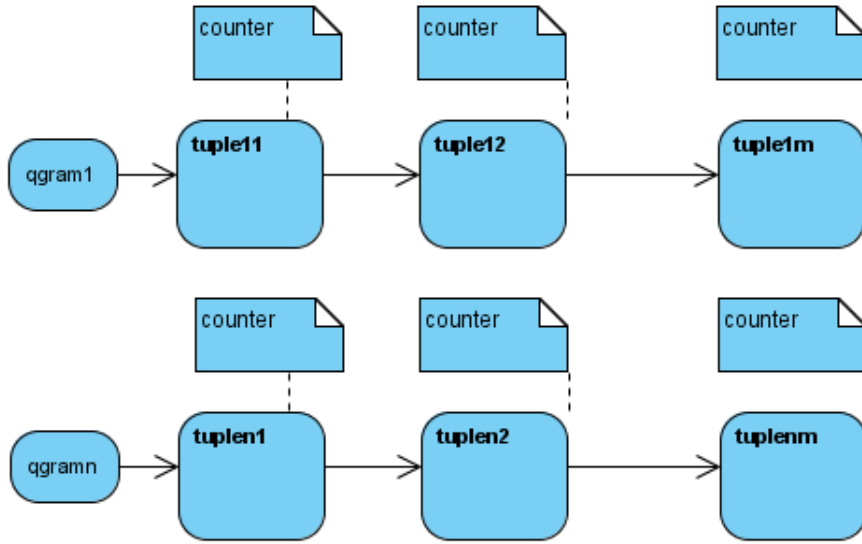


Figure 3.3: The qgram hash table

introduced in the *iterator* model, which had no parameters at all (see section 3.1). We still reassure the readers that this version of the algorithm is here represented only for clarity reasons. In order to obtain the same behavior, the input operators could be passed once for all during the *open* method computation and some simple tricks could further be used inside a *next* method without parameters. A second simplification was that of presenting the *next* method as a recursive one, for clarity reasons again. The real implementation of the algorithm uses an infinite loop instead, in order to be computationally more efficient in real executing environments (maybe cite a book over compilers and interpreters).

As stated before, the idea behind the algorithm is to use the best characteristics of *ssjoin* and *shjoin*, melting the approximate matching approach from the first with the incremental computation from the second. Intuitively the algorithm works in the following way: At every execution step a new tuple is read in turn from one of the inputs, let it be *insert*. The qgram hash table of *insert* is then updated with several new entries, all pointing to the new tuple and each corresponding to a particular qgram contained in its join attribute. The hash table of the other input, let it be *probe*, is then probed to find tuples whose join attribute is *similar* to the currently read one and all matches are returned in sequence. It is important to know that the hash table of *probe* contains only the tuples that have already been read from this input operator, and not those still to be read. After all matches have been returned the algorithm symmetrically performs the same actions inverting the input operators: A new tuples is read from *probe*, its corresponding hash table is updated with new entries and the *insert* hash table is probed for matches. This procedure is repeated until both input operators have been exhausted (no more tuples can be read from any of them) and all possible matches have been returned.

Before entering in the details of the algorithm let's first explain some necessary notions. For the symmetric hash join algorithm a quiescent state is one in which a new tuple has to be read from one of its inputs and all matching results to this point have been returned. The same principle continues to hold for our symmetric set hash join algorithm. While entering a quiescent state a new tuple must be read and some internal variables must be initialized. The most important among the variables is the *iterator* initialized in line 21. This variable is initialized at the beginning of the current quiescent state and its value is maintained by the algorithm till the next quiescent one, it doesn't matter how many times the *next* method has been called in the meanwhile. This means that if the *next* method is called consecutively twice and at most only in the first time the algorithm has entered a new

---

```

Input: insert:Operator, probe:Operator
Output: a new matching tuple
1 if isQuiescent() then
2   if insert.hasNext() then
3     current  $\leftarrow$  insert.next();
4   else
5     if probe.hasNext() then
6       return next(probe, insert);
7     else
8       return null;
9     end
10  end
11  end
12 end
13 setQuiescent(false);
14  $\{q_1, \dots, q_n\} \leftarrow$  qGrams(current);
15 insert.updateHashTable( $\{q_1, \dots, q_n\}$ , current);
16 probe.initializeCounters();
17 foreach  $q \in \{q_1, \dots, q_n\}$  do
18    $\{t_1, \dots, t_m\} \leftarrow$  probe.getTuplesFromQGram(q);
19   probe.incrementCounter( $t_1, \dots, t_m$ );
20 end
21 iterator  $\leftarrow$  probe.getTupleCounterIterator();
22 end
23 while iterator.hasNext() do
24   tc  $\leftarrow$  iterator.next();
25   if tc.counter  $\geq k$  then
26     return current  $\cdot$  tc.value;
27   end
28 end
29 setQuiescent(true);
30 return next(probe, insert);

```

Figure 3.4: The next method

---

quiescent state, the value of the iterator at the beginning of the second call will be the same of that at the end of the first.

*insert* and *probe* are the names of the two join input operators. Remember that in the symmetric hash join tuples are read from both operators in turn. This means that every time a new tuple is read from one of the operators (the *insert* one) it is stored in its corresponding hash table and the other operator hash table (*probe* one) is probed for matches. We assume that the input operators are pipelined as in Kwanchai's architecture [EFP06]. This means that the operators are composed by, at least, the four methods below

1. *open*()
2. *next*()
3. *close*()
4. *isQuiescent*()

Our *sshjoin* operator follows the same architecture as well, but only the algorithm of the method *next* is given here, the others are straightforward. When the *open* method is called at the beginning of the algorithm, the two qgram hash tables together with the *iterator* and *k*, the threshold value discriminating between a match and a non match, are initialized. The algorithm state value is set to *quiescent*. The *next* method is then iteratively called until all join results have been returned. At the end, by calling the *close* method the space utilized by the two hash tables and the *iterator* is released. The input operators have some further important methods that implement part of their particular behavior

**setQuiescent**(boolean *b*) sets the operator quiescent state value to *b* (*true* or *false*).

**updateHashTable**(Set qgram  $\{q_1, \dots, q_n\}$ , tuple *t*) the hash table of the operator is updated with *n* new entries, one for each qgram contained in the qgram set. All entries point to the same tuple *t*.

**initializeCounters**() all counters in the operator hash table are initialized to 0.

**getTuplesFromQGram**(qgram *q*) this method returns all tuples in the operator hash table containing qgram *q* in their join attribute.

**incrementCounter**(tuple *t*) the counter corresponding to tuple *t* in operator hash table is incremented by 1.

**getTupleCounterIterator**() an iterator over the tuples in the operator hash table is returned, each entry consisting of a composed *TupleIterator* object containing two fields

**value** the value of the tuple,

**counter** the counter associated to the corresponding tuple in the operator hash table.

In addition, an auxiliary method is used to simplify the algorithm explanation

**qGrams**(tuple *t*) returns a list of all qgrams contained in the join attribute of *t*.

Now we are ready to describe the algorithm behavior in detail. The recursive calls to *next* in line 6 and 30 are used for the algorithm to continue the join procedure until a new match is returned. In order for the algorithm to be symmetric, meaning to continually read from one of the inputs while probing the other in turn, the *insert* and *probe* operators are appropriately chosen in the following way

1. At the algorithm beginning, the *insert* operator is the left one and the *probe* the right,
2. Supposing that the last reading operation from the join inputs was made from *insert*, during the new read the *old insert* is going to be the *new probe* and the *old probe* the *new insert*.

In figure this behavior is achieved by calling the *next* method with inverted parameters with respect to the call in process.

In line 3 a new tuple is read from the *insert* operator if this has not yet been exhausted. If the operator has no more values to produce on the other hand, a test over the second join input is performed in lines 4–11. In case this is not empty the *next* method is recursively called with inverted parameters, reading the new tuple from the non-empty input. On the other hand if both inputs are exhausted the algorithm returns a *null* value (line 8), meaning that the join execution had arrived to its end.

The state of the operator is consecutively set to *not quiescent* in line 13.

After the new tuple has been read, the qgrams of its join attribute are obtained in line 14 and the qgram hash table of the *insert* operator is updated in order to consider this new entries (line

Name	Mansion	Department
Tom Smith	engineer	R&D
Will Smith	salesman	Sails
John Cusack	economist	Marketing
Ray Blue	salesman	Sails

(a) Employees

Name	Num Employees	Budget
Sailes	100	1000
Marketing	30	800
R&D	10	500

(b) Departments

Figure 3.5: Tables

15). More precisely, for every qgram  $q$  in the join attribute of  $current$ , the hash value of  $q$ ,  $h(q)$ , is retrieved and the  $current$  tuple is stored in the entry corresponding to  $h(q)$  key in the hash table.

In lines 16–20 the similarity between the  $current$  tuple and the ones stored in the  $probe$  hash table is computed using the  $counter$  field previously introduced. First of all, the value of the field is set to 0 for all tuples in the  $probe$  hash table (line 16). Then, for every qgram  $q$  of the  $current$  tuple join attribute, all tuples containing the same qgram in their join attribute are retrieved and their  $counter$  is incremented by one (line 19). At the end of the loop the  $counter$  field of every tuple in  $probe$  will contain the number of qgrams in common with  $current$ .

In line 21 an iterator over all tuples in  $probe$  is returned, together with the  $counter$  value for each tuple.

In lines 23–28 all tuples in the above returned iterator are probed to find those similar to  $current$ . Notice that the  $iterator\ next$  method is called in line 24. This method has nothing to do with the  $sshjoin\ next$  method and is used to obtain in order all tuples contained in the  $iterator$ . In the algorithm, the intersection measure between the corresponding qgram sets is used (line 25), but other similar measures could be used as well (e.g. normalized intersection, weighted intersection etc). Only if this intersection is greater than a given threshold, i.e.  $k$ , then a new valid match is returned concatenating the  $current$  tuple with the one retrieved from the  $probe$  hash table (line 26). If no more tuples can be read from the  $iterator$ , the algorithm enters a new quiescent state, setting the quiescent state value to  $true$  in line 29 and recursively calling  $next$  with inverted parameters in line 30.

Let’s now illustrate how the symmetric set hash join algorithm works in the following example.

**Example 13.** Suppose that the qgram size used by the algorithm is 3 and the intersection threshold between qgram sets is  $k = 2$  (tuples having an intersection of 2 or more between their join attribute qgram sets are the only valid matches). Suppose also that the left join operator is a scan operator over the Employees table and the right one is another scan operator over Departments table (see Figure 3.5). The scan operators are implemented using the pipelined architecture introduced in section 3.1 and read tuples from their respective tables one at a time. At the algorithm beginning Employees is treated as the input operator and the other one as the probe. Tuple [Tom Smith, engineer, R&D] is read first (this is the value of  $current$  in the algorithm). The qgram set of this tuple join attribute is obtained: {R&D}. The tuple is stored in the Employees hash table as its only entry

1. R&D  $\longrightarrow$  [Tom Smith, engineer, R&D]

The hash table associated to Departments is still empty, so the probing procedure ends very quickly.

The scan over table Departments is now treated as the insert operator and tuple [Sailes, 100, 1000] is read as a new value. The qgram set of the tuple is: {Sai, ail, ile, les}. As a consequence four new entries are stored in the Departments hash table

1. Sai  $\longrightarrow$  [Sailes, 100, 1000]
2. ail  $\longrightarrow$  [Sailes, 100, 1000]
3. ile  $\longrightarrow$  [Sailes, 100, 1000]

4. les  $\longrightarrow$  [Sailes, 100, 1000]

As we can notice, all the entries contain a pointer to the same tuple [Sailes, 100, 1000]. The Employees hash table is probed for matches afterwards. The only entry in this table is R&D  $\longrightarrow$  [Tom Smith, engineer, R&D]. Notice that qgram R&D is different from all qgrams contained in the current tuple's join attribute. As a consequence the counter value of this entry will be set to 0 and no match will be returned.

Tuple [Will Smith, salesman, Sails] is now read from the Employees table by its scan operator (the new input). The corresponding join attribute's qgram set is retrieved: {Sai, ail, ils} and three new entries are stored in the hash table, which will now contain four entries in total (the old entry plus the three new ones):

1. R&D  $\longrightarrow$  [Tom Smith, engineer, R&D]

2. Sai  $\longrightarrow$  [Will, Smith, salesman, Sails]

3. ail  $\longrightarrow$  [Will, Smith, salesman, Sails]

4. ils  $\longrightarrow$  [Will, Smith, salesman, Sails]

The counter values in the Department hash table are recomputed for every tuple. At the end of the procedure the tuple [Sailes, 100, 1000] will have a counter value of 2 corresponding to the two qgrams in common with the current tuple

1. Sai

2. ail

As the threshold value is set to 2, the tuple is returned as a match, even its join attribute value is not exactly equal to Sailes.

Tuple [Marketing, 30, 800] is now read as the current tuple. The corresponding qgram set list is retrieved: {Mar, ark, rke, ket, eti, tin, ing}. The qgram hash table of Departments will now contain

1. Sai  $\longrightarrow$  [Sailes, 100, 1000]

2. ail  $\longrightarrow$  [Sailes, 100, 1000]

3. ile  $\longrightarrow$  [Sailes, 100, 1000]

4. les  $\longrightarrow$  [Sailes, 100, 1000]

5. Mar  $\longrightarrow$  [Marketing, 30, 800]

6. ark  $\longrightarrow$  [Marketing, 30, 800]

7. rke  $\longrightarrow$  [Marketing, 30, 800]

8. ket  $\longrightarrow$  [Marketing, 30, 800]

9. eti  $\longrightarrow$  [Marketing, 30, 800]

10. tin  $\longrightarrow$  [Marketing, 30, 800]

11. ing  $\longrightarrow$  [Marketing, 30, 800]

The Employees hash table will be probed and counter values updated. As any of the new qgrams is a valid entry in this hash table any tuple will have a counter value bigger than 2 and no match will be returned.

The computation will continue this way until both join inputs have been exhausted. At the end the join result will be composed of the following matches:

1. [Will, Smith, salesman, Sails ,Sailes, 100, 1000] *similarity* = 2
2. [Ray, Blue, salesman, Sails, Sailes, 100, 1000] *similarity* = 2
3. [John, Cusack, economist, Marketing, Marketing, 30, 800] *similarity* = 7.

## 3.4 Algorithm analysis

In this section a detailed analysis of the algorithm computational and spatial complexity is given confronting it with the normal symmetric hash join complexity. At the end some improvements that further reduce its overall complexity are mentioned.

### 3.4.1 Computational analysis

The sshjoin algorithm is a pipelined operator that follows the *iterator* architecture described in section 3.1. First of all this means that the algorithm is formed of at least the methods *open*, *next*, *close* and *isQuiescent*. The *open* and *close* methods are called only once, respectively at the beginning and at the end of the query evaluation and their only useful work consists in initializing and deallocating some data structures. As these operations performance depend on the particular hardware used and they are called only once in the whole operator life cycle, their computational complexity is ignored. The *next* method on the other hand is iteratively called every time a new result must be returned and is though the heart of the entire algorithm. The *isQuiescent* method is called whenever necessary from the adaptive component to test the internal state of the sshjoin operator. The complexity of this further method is constant because it only has to perform a simple test over the state of an algorithm internal variable. In the following we will concentrate on the *next* method analysis supposing that  $n$  values have been read from both operators, meaning that both hash tables contain  $n$  tuples, and that the average size of the join attribute is  $|jattr|$ .

Lets first estimate the computational complexity of the methods called inside *next*. The method is intended to be called inside a pipelined architecture composed of other operators as well. This means that the method will call the *iterator* architecture methods of its input operators in the pipeline. The computational complexity of this methods depends on the particular type of physical operator and can highly vary from a constant complexity in the case of a *Scan* operator to a quadratic complexity in the case of a *nested loop* join. We will though ignore the complexity of such methods in our analysis.

As stated before, the *isQuiescent* method in line 1 has a constant computational cost. The only thing this method does is test if an internal boolean variable value is *true* or *false*.

The method *setQuiescent()* in line 13 has also a constant cost as it only sets the value of an algorithm internal variable.

Method *qGrams(current)* in line 14 has a complexity linear to the join attribute length. In order to obtain all qgrams of a given join attribute value  $jatt_i$ , a linear scan is necessary and  $|jatt_i| - q + 1$  different qgrams are obtained, where  $q$  denotes the qgram size used by the algorithm.

In line 15 all entries corresponding to the previously obtained qgrams are updated. The cost of a single update is constant as the new tuple is concatenated to the end of the entry list (see Figure 3.3). The total operation cost is though linear to the number of qgrams and as a consequence to the length of the join attribute value.

Method *initializeCounters* in line 16 has cost  $n$ , linear to the current number of tuples contained in the operator hash table.

Method *getTuplesFromQGram(q)* in line 18 has a complexity linear to the dimension of a single entry bucket in the qgrams hash table. Let this dimension be approximately logarithmic to the size of the whole hash table of *probe*. If  $n$  values have been read from the *probe* operator this quantity will be  $\log(n)$ .

Every single call to method *incrementCounter* in line 19 has a constant cost as it only increments a specific counter.

Let method *getTupleCounterIterator()* in line 21 have a constant cost. If a linked list of pointers to all tuples in the hash table were maintained, then this assumption would surely hold. A pointer to the top of the list should be returned as the new *iterator*.

Methods *hasNext* and *next* of the *iterator* together with concatenation operation  $*$ , in line 26, have constant cost if implemented appropriately.

Now we are able to estimate the complexity of a single step of the *next* method, meaning the time between two consecutive quiescent states. Inside a quiescent state lines 1–22 are going to be called only once. Every iteration of the *ForEach* structure in lines 17–20 has an average cost of  $\log(n)$ , the logarithm of the number of tuples contained in the operator hash table. As the *ForEach* is repeated a number of times linear to  $|jattr|$ , the join attribute average length, the overall complexity of this lines is  $|jattr| * \log(n)$ . Ignoring the complexity of the input operators *iterator* methods, the complexity of these lines is in  $\theta(n + |jattr| * \log(n))$ , where  $|jattr|$  is the average length of the join attribute and  $n$  the number of tuples read from the actual *probe* iterator. The *while* in lines 23–28 is going to be called once for every tuple contained in the *probe* operator hash table, meaning  $n$  times according to the above assumption. The cost of the single *while* iteration is going to be constant. Line 29, finally, has a constant cost.

In conclusion, the overall cost of the *next* method single step after  $n$  values has been read from *probe* operator is going to be in  $\theta(n + |jattr| * \log(n)) \approx \theta(n)$ , if the average size of the join attribute is a lot smaller than  $n$ ,  $|jattr| \ll n$ . Suppose that the cardinality of the first operator is  $M$  and that of the second  $N$  such that  $N \leq M$ , without lose of generality. Then the computational cost of the algorithm after having read the first  $N$  values from both join operators will be

$$2(1 + 2 + \dots + N) = 2 \frac{(N - 1)N}{2} \in \theta(N^2) \quad (3.1)$$

At every step the complexity is linear to the number of values read from the *probe* operator, beginning with 1 at the first and ending with  $N$  at the last. As readings are made from both operators, the 2 factor at the beginning of the operation is used.

For the resting  $M - N$  readings from the operator with the bigger cardinality, every reading will cost  $N$ . The cost of the remaining computation will so be

$$(M - N) * N \quad (3.2)$$

The whole computational complexity is though in  $\theta(N^2 + (M - N) * N)$ , meaning quadratic in the size of its inputs. Notice that if both inputs have the same cardinality,  $M = N$ , the complexity would be in  $\theta(N^2)$

### 3.4.2 Space complexity analysis

Having estimated the overall computational complexity of the algorithm let's now have a look at its spatial complexity, meaning the memory needed by the algorithm during its execution. Suppose again that the cardinality of the left operator is  $M$  and that of the right one  $N$ , such that  $N \leq M$  and that  $n$  values have been read from both operators in a generic algorithm step. Let  $|jattr|$  be the average size of the join attribute,  $t$  the average size of a tuple and  $q$  that of the single qgram.

Remember that the qgram length is fixed and constant. The memory occupied by the algorithm to this point is that used by the operators to maintain the tuples in the respective hash tables. Figure 3.3 suggests that for every qgram a list of tuples containing that qgram in their join attribute should be maintained. The disadvantage of this method is that the same tuple is going to have more instances in memory, exactly one for every qgram present in its join attribute, resulting in a bad memory management. The problem could be resolved by maintaining in main memory a single instance for every read tuple and buckets of pointers to tuples, instead of buckets made by tuple values as reported in Figure 3.6.

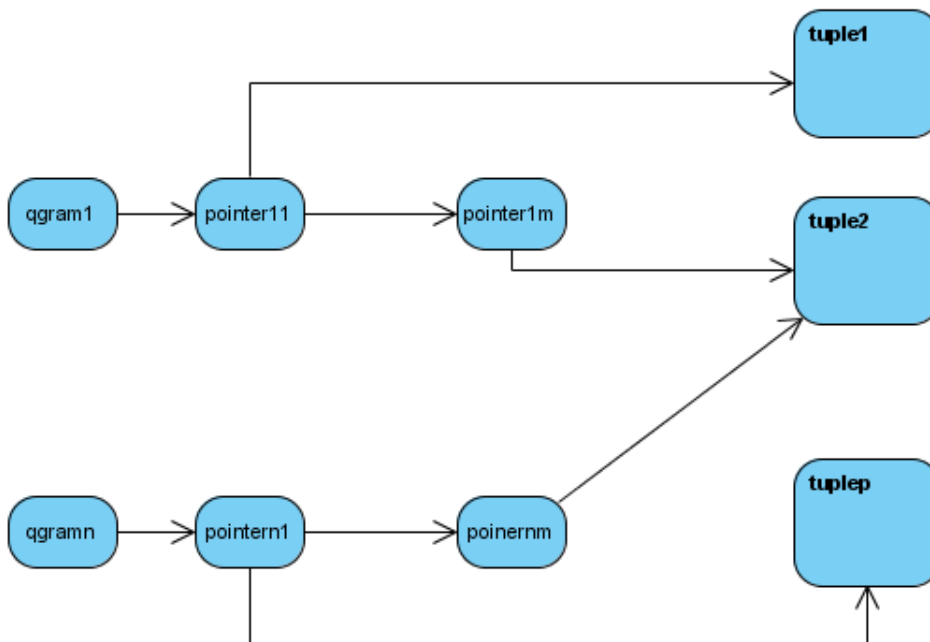


Figure 3.6: The qgram hash table

As the space required for the pointers is normally less than that required by a tuple the overall space complexity of the algorithm results improved. If we denote with  $|pointer|$  the space occupied by a single pointer, then the space required by the algorithm after reading  $n$  values from both operators is that for the respective tuples, pointers and qgrams

**tuples**  $2nt$ , there are  $2n$  tuples read

**pointers**  $2n|jattr||pointer|$ , there are  $2n$  tuples read, every tuple having a number of qgrams linear to its join attribute length and every qgram in the hash table points to its respective tuple

**qgrams** at most  $2nq|jattr|$ , if all qgrams were different

In conclusion the overall space complexity of the sshjoin algorithm at step  $n$  is in  $O(2n(t + |jattr|(q + |pointer|))) \in O(n)$ , as all the other parameters are constants. At the end of the join computation the space required by the algorithm would be in  $O(M + N)$ , linear in the size of its inputs.

### 3.4.3 Comparison with symmetric hash join

Now that we have estimated the computational and spatial costs of the newly introduced symmetric set hash join (sshjoin) algorithm let's make a confront with the respective costs of the normal symmetric hash join algorithm.

	shjoin	sshjoin
Obtain qgrams	$ jattr  + q - 1$	0
Update hash table	$ jattr  + q - 1$	1
Calculate counter values	$( jattr  + q - 1) * B$	0
Find matches	$( jattr  + q - 1) * B$	$B$
Total cost	$2( jattr  + q - 1)(1 + B)$	$1 + B$

Figure 3.7: shjoin-sshjoin costs confront

It is well known in the literature that the computational cost of a normal hash join is log-linear in the size of its inputs,  $O(N \log(N))$ . At a high level we can think at the symmetric hash join algorithm as a simple hash join performed twice

1. The second operand is entirely read and hashed, then the first is read in sequence probing the second operator hash table
2. The first operator is entirely read and hashed, then the second is read in sequence probing the first operator hash table

Even though performing a simple hash join twice is redundant, it is necessary a single computation for the join to produce full results, it gives the idea of a symmetric hash join in terms of a more well known algorithm and gives an upper bound to the shjoin computation cost in  $O(N \log(N))$ , where  $N$  is the maximal cardinality of the join operators. In conclusion we can see that there is a big gap between the complexity of shjoin,  $N \log(N)$ , and that of a sshjoin,  $N^2$ , resulting in a better performance of the first with respect to the second. On the one hand, the complexity of the second is intrinsically quadratic in the size of its inputs as an approximate join has to compare every single pair of tuples belonging to the different operators. On the other hand, the sshjoin result quality is better as it returns all approximate join results when necessary, in addition to the exact join ones, resulting really convenient in those scenarios where data quality may be poor.

In addition to the overall complexity confront between shjoin and sshjoin, we believe it is convenient to give an estimation of how more efficient a single step, meaning the computation between two consequent quiescent states, of the shjoin is in confront with our sshjoin. In the following  $|jattr|$  will denote the average length of the join attribute,  $q$  the qgram length,  $B$  the average length of a single bucket in the hash table. We suppose that buckets in both algorithms have the same average length, even though intuitively qgram buckets tend to be bigger than attribute one because several attributes may share a common qgram.

First of all we estimate the costs of 4 principal operations for both algorithms (see Figure 3.7)

**Obtain qgrams** this operation is needed only for the sshjoin algorithm and is performed a number of times equal to the number of qgrams contained in the join attribute,  $|jattr| + q - 1$ . Every single qgram is obtained with a constant cost.

**Update hash table** this operation is performed a number of times linear to  $|jattr| + q - 1$  for sshjoin and only once for the shjoin. Every single update has a constant cost is implemented appropriately.

**Calculate counter values** This operation is performed only for the sshjoin algorithm. For every qgram contained in the join attribute the corresponding bucket is obtained and the contained tuples counter value is incremented. As there are  $(|jattr| + q - 1)$  interested buckets, every bucket contains an average of  $B$  tuples and every single increment operation has constant cost, the total cost of the operation is  $(|jattr| + q - 1) * B$ .

**Find matches** If appropriately implemented the sshjoin, at least all tuples which counter has been incremented by at least 1 in the previous operation should be probed for matches to see if the counter value is higher than a certain  $k$ . The cost of a single counter check is constant, as a consequence the total cost of this operation for the sshoin is  $(|jattr| + q - 1) * B$ , equal to the cost of the counter increment in the previous operation. In the shjoin a single bucket is probed. Supposing each tuple join attribute confront as constant, the cost of the operation is equal to the size of a bucket  $B$ .

We believe that this operations are the computationally most signifying for the algorithms, meaning that the other operations computational cost is irrelevant compared to this. At the end the *Total cost* is estimated as the sum of the above costs being

- $2(|jattr| + q - 1)(1 + B)$  for the sshjoin,
- $B + 1$  for the shjoin.

The ratio sshjoin step - shjoin step is so

$$2(|jattr| + q - 1) \tag{3.3}$$

meaning that every step of the sshjoin costs  $2(|jattr| + q - 1)$  times the cost of a shjoin. In reality this ratio is lower than the real one because of the convention used during our analysis that the average length of a qgram bucket was equal to that of an attribute one. Experiments could be performed to obtain real numbers for having more precise results over the buckets size and as a consequence more precise cost. Still we believe that for the purpose of this thesis, the estimated cost is enough even though it is an upper bound to the real one.

The following example illustrates in a real scenario how the cost of sshjoin single step can be more expensive than a shjoin single one.

**Example 14.** *Suppose that the average size of a join attribute is of 15 characters,  $|jattr| = 15$ , which can be true if the attribute represents people full names. Let also the qgram length be 3, a commonly used convention. A single step of sshjoin is then going to be  $2(|jattr| + q - 1) = 2(15 + 3 - 1)$ , 34 times more expensive than a shjon single step.*

*At the end, the spatial complexity of the sshjoin algorithm is of the same order of the shjoin one, being both in  $O(M + N)$ , where  $M, N$  are the cardinality of the join input operators. The difference is that the sshjoin uses more space to maintain the qgrams and more pointers for bigger buckets, as explained in section 3.4.2.*

## 3.5 Pros and cons

The symmetric set hash join algorithm seems to be perfect so far, but as any algorithm there are advantages and disadvantages in using it. Lets now see more in detail these aspects of the algorithms. The algorithms main advantages are

1. sshjoin is an incremental operator, suitable for use in a pipelined architecture.
  - (a) Initial response time of the algorithm is drastically reduced, meaning that results can potentially begin to be returned immediately at the execution beginning.
  - (b) The algorithm could be easily integrated in a pipelined architecture.
  - (c) The algorithm is suitable for stream join scenarios in which information about the stream data is never complete, but incrementally obtained

2. The algorithm structure is compact and simple (see algorithm in Figure 3.4). As a consequence its implementation and testing does not require a particularly high effort.
3. The computational efficiency of the algorithm is not bad and by little modifications can be further improved.

The algorithm main disadvantage is

1. The algorithm requires big amounts of main memory to store its partial hash tables.

As we see the only problem with the algorithm is the big quantity of space required. Precisely every tuple read from its inputs, namely  $R$  and  $S$ , has to be stored in one of the two hash tables. At the beginning of the algorithm though no space is required, but every time a new tuple is read from the inputs it might be stored in the corresponding hash table. At the end of the algorithm the required space will be  $|R| + |S|$ . If at some execution point this required space is higher than the available main memory then the algorithm is not applicable and further improvements should be made (e.g. intelligently swap to disk not necessary tuples currently maintained in main memory by using Last Recently Used algorithms). In addition the algorithm requires further space for the qgrams and pointers used in the hash table buckets, as mentioned in 3.4.3, resulting in a worse usage of memory than a normal shjoin.

# Chapter 4

## Quality aware adaptive query processing

The duplicates matching, or object identification, problem is the most dealt with data quality problem in literature. In this chapter an introduction to the general problem is given first. As the problem is too big and intricate we choose a particular subproblem and give a general solution to it. The rest of the chapter is organized as follows: Section 4.1 introduces the general data quality problem dealt with in this thesis. In section 4.2 necessary definitions used in the remainder of the thesis are introduced. Section 4.3 provides a high level solution to the previously introduced problem. Finally, in section 4.4, the specific problem solved in this thesis, as part of the more general one introduced in section 4.1, is presented.

### 4.1 Initial problem specification

In this section the duplicate matching problem is described first. Then, the particular subproblem of online duplicate detection is introduced as the subject of main interest in the rest of this thesis. A general approach to the solution is given together with a validation method to estimate its efficiency.

#### 4.1.1 Online duplicate matching: (Stream) Joins

A *duplicate* is a value that represents a real life entity, but suffers from accuracy problems (e.g. misspelling errors). For example, a duplicate for *Roald Lengu*, the author of this thesis would be *Roal Lengu*. Note that in the latter case a *d* is missing in the author's name. Databases and datawarehouses often contain a certain degree of duplicates because of one or more of the following reasons [EIV07]

1. transcription errors
2. missing information
3. lack of standard formats

The *duplicate matching* problem, introduced in section 1.3, under the name *object identification problem* is probably the most important and most extensively investigated data quality activity [BS06]. Given two values  $v_1$  and  $v_2$ , the goal of this method is to understand if the values correspond to the same real world entity even if their representations were different ( $v_1 \neq v_2$ ). Several approaches have been proposed to solve the problem, but according to Bilenco et al. none of these is universally optimal [BMC<sup>+</sup>03]. Still the majority concentrates on solving the duplicate matching problem inside a single table. This functionality can be thought as that of a filter that takes in input a table with duplicates and returns in output the same table without duplicates. Some other methods, like similarity set join (ssjoin) [CGK06], solve the duplicates matching problem during join execution

between two or more tables. These particular types of joins are called *approximate joins* and the duplicate detection in this case is said to be *online*.

The main disadvantage of approximate joins is their high computational complexity. If in many cases the computational complexity of a *normal join* could be reduced to linear (e.g. merge join) or log-linear (e.g. hash join) with respect to the tables sizes, the computational complexity of an approximate join is intrinsically quadratic. A higher computational complexity means higher execution time and in many real life situations this could be problematic.

Just think about the scenario of an approximate join between streams in which every stream produces  $n$  values per second. As the streams are potentially infinite, and a join between infinite streams is impossible, a (temporal) window of finite size  $N$  (seconds) over the streams is defined. The join should though be calculated between the first  $nN$  values of both streams, than on the second  $nN$  and so on, transforming an infinite join in an infinite series of finite joins. In  $N$  seconds though the streams would produce  $nN$  different values. If the time to calculate the approximate join between these  $nN$  values were higher than  $N$  seconds (e.g.  $N + k$ ) then the expected results would arrive out of time by  $k$  seconds. Suppose now that these results were used to control real time extremely sensible parameters in a nuclear reactor. If the value of  $k$  was too high then controlling the parameters would be impossible and a disaster could happen.

The above example points out that an approximate join is not suitable for stream joins, in which results should be calculated on the fly and in which time constraints are imposed.

Another problem of approximate joins is that, to our knowledge, there does not exist an *incremental approximate join* algorithm for a stream scenario. In this kind of scenarios the information about values contained in at least one of the streams is not complete till the end of the join execution (over a window if the stream is infinite). There is therefore need for a join algorithm that, according to the more complete information gained step by step over the streams, incrementally begins to calculate the complete execution result set. Suppose that, in the stream based scenario above two types of joins were used

1. a non incremental join that begins calculation only after an entire window has been read
2. an incremental join that produces  $\alpha x$  values, for some coefficient  $\alpha$ , after  $x$  values have been read from each stream

In the first case after  $N$  seconds still no results would be returned. In the second one after  $N$  seconds  $\alpha N$  values would have been already calculated and would be ready for further use (to control the important parameters in the case of the nuclear central). The property of calculating results even with partial information could be particularly interesting in a pipelined architecture, in which partial results of previous operators could be used from further ones (see Figure 4.1).

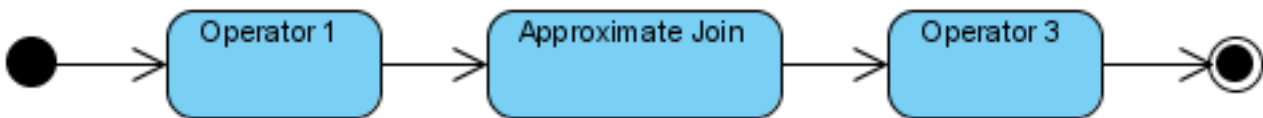


Figure 4.1: A pipeline architecture example

### 4.1.2 General approach to the solution

As previously discussed in order to have an efficient *online duplicate detection* approach the following problems should be resolved

1. Provide an incremental approximate join algorithm

## 2. Overcome the intrinsic quadratic complexity of this approximate join

Supposing to use the sshjoin algorithm introduced in chapter 3, suitable for stream based scenarios, the second problem continues to hold and no solution appears to be feasible. In order to guarantee that no matches are lost during an approximate join, any tuple from the first table should be compared with all tuples from the second one. As a consequence any algorithm that makes such a guarantee, even our sshjoin, has an intrinsic quadratic complexity and nothing can be done to reduce it.

The only way to solve the complexity problem is to relax the above condition on the soundness of the join computation. Instead of using an approximate join during the entire execution, the idea is to use the approximate join only when it is necessary because *too many* matches are missing and use the normal join otherwise. As a result, at the end of the join execution an acceptable trade-off between the number of missed matches (quality of returned data) and the total execution time (computational complexity) would be achieved.

The idea is though, to construct a system that continually oscillates between two different states:

1. Normal join execution
2. Approximate join execution (sshjoin)

according to the quantity of matches currently being lost. In Figure 4.2 the system state machine diagram is given. Some effort is necessary to define the *guards* in the transition arrows, in order for the system to be in the right state at the right time.

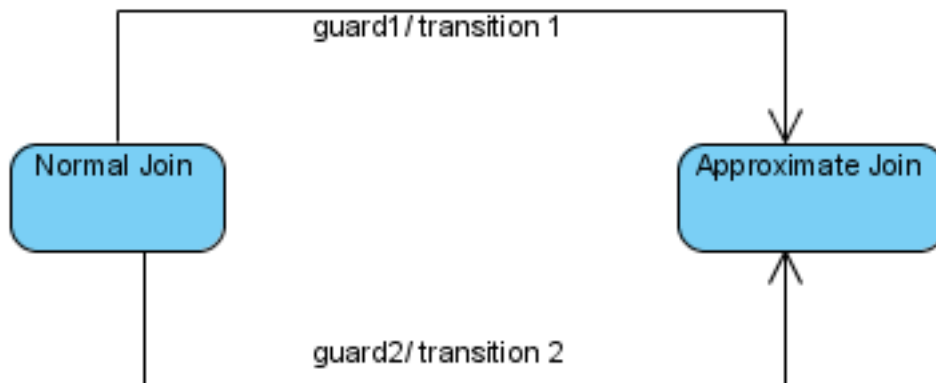


Figure 4.2: System state machine diagram

### 4.1.3 Approach validation

Once the general approach to the problem has been defined, there is the need to compare its behavior with both traditional methods

1. a normal (exact) join performed for the entire execution time,
2. an approximate join performed for the entire execution time.

Intuitively, in case a normal join were used during all the computation, some semantically correct, but syntactically incorrect matches would be lost, where with semantically correct we mean tuples representing the same real-life entity but that probably are not the same according to the standard string equality function, while with syntactically correct we mean tuples having the same

representation according to the string equality function. The advantage of the normal join in this case would only be that of having lower response times.

On the other hand, performing a complete approximate join algorithm during the entire computation would intuitively result in a more complete result set, as many semantically correct matches but syntactically incorrect ones would be returned, but in the same time in higher response times. The usage of an entire approximate join is not justified either in those cases in which duplicates are not present at all, or their presence rate is so low that missing a few matches could be tolerated with respect to a great gain in response times.

According to these two intuitions we use two indexes to report the effectiveness, as measure of the completeness of the result, and efficiency as measure of the response times low levels.

$$Effectiveness = \frac{RS_{ap} - RS}{RS_{ap} - RS_{ex}} \quad (4.1)$$

where  $RS$  is the join result size of our approach,  $RS_{ap}$  the result size of an entire approximate join, and  $RS_{ex}$  the result size of an entire exact join. Intuitively the effectiveness index measures the number of approximate matches correctly returned by our approach, considering no approximate matches would be returned by an entire exact one and the maximum possible number of returned matches as those returned from an entire approximate join. High values of the index show good effectiveness of the system.

$$Efficiency = \frac{RT_{ap} - RT}{RT_{ap} - RT_{ex}} \quad (4.2)$$

where  $RT$  is the response time of our approach,  $RT_{ap}$  the response time of an entire approximate join, and  $RT_{ex}$  the result size of an entire exact join. Intuitively the efficiency index measures the response time of our approach, considering the lower possible the response time of an entire exact join execution and the highest possible the one of an entire approximate join execution. Lower values of this index would show good efficiency of the system.

## 4.2 Some definitions

Now we are ready to introduce some important definitions used in the following of this thesis. All the definitions share the same scenario: A symmetric set hash join  $R \bowtie S$ , between tables  $R$  and  $S$  is being executed.

In the following, when we refer to a set we mean a set with repetitions i. e. a bag, if not explicitly stated otherwise. For example  $\{\text{Roald}, \text{Marco}, \text{Roald}\}$  is a bag, while the corresponding set will be  $\{\text{Roald}, \text{Marco}\}$ .

In the following, terms *table* and *stream* are used as synonyms if not explicitly stated otherwise (e.g., in a stream based scenario), meaning a join input.

**Definition 1.** Given tuple  $t$  from any of the tables involved in the join the functions  $jAttr : Tuple \rightarrow String^n$  associates with  $t$  the value of its  $n$  join attributes.  $jAttr(t) = \text{join attributes of } t$ .

In Example 13  $jAttr([\text{Marketing}, 30, 800]) = \text{Marketing}$ .

As an incremental operator the symmetric set hash join has only partial vision of its inputs during all the execution and the returned results will so depend on the vision dimension. The  $\text{sshjoin} \bowtie$  can though be thought as a dynamic operator having an internal state that changes over time, a convenient notation to represent this state is required though.

**Definition 2.** A join execution point is a time instant in the join algorithm execution, in which the next operation to be carried out is the reading of a new tuple from table  $S$ . The notation  $\bowtie_n$  is used to refer to the join operator  $\bowtie$  whose execution is stopped at point  $n$ .

From the definition, at the  $n^{\text{th}}$  execution point of join  $\bowtie$ ,  $n$  values have been read from both tables  $R$  and  $S$ , all matching tuples at this point have been returned and the join is ready to read a new tuple from table  $R$ .

**Definition 3.** If join  $R \bowtie S$  is in execution point  $n$  and table  $R$  has a foreign key constraint in table  $S$ , then the set of foreign keys read so far is denoted by  $FK(n)$  and the set of primary keys read by  $PK(n)$ .

The definition above introduces some important notions in the case of a foreign key join, which is also the most common type of join in practice.

As previously mentioned the symmetric set hash join algorithm has only knowledge of the old read data. If for any reason we are interested in estimating the characteristics of the new data still to be read then we should make use of the information contained in the old data. Intuitively a single join input stream could be thought as a contiguous series of horizontal regions (made of a certain number of tuples), each of which has its own particular *local characteristics* different from that of its neighbors. Using this concept we can conclude that the characteristics of data still to be read are similar to those of currently read data (we are continuing to read data from the same horizontal partition). In Figure 4.3 an input stream composed of four different zones is given. Notice that the length of these partitions can be different as well. The two extreme cases will be

1. a single partition stream,
2. a stream made by very small horizontal partitions (nearly a tuple for partition).

A stream made of a single partition is also called a *homogeneous stream*. In contrast a stream made by more than one partition is called a *heterogeneous* one.

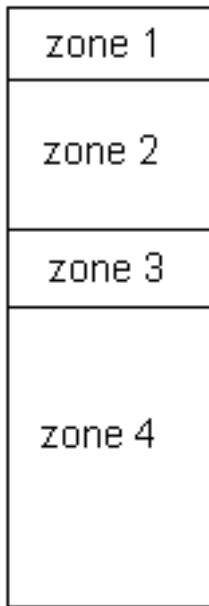


Figure 4.3: Horizontal zones in a data stream

Lets now give some the formal definitions of some local data characteristics.

**Definition 4.** The percentage of duplicates  $dp$  in a partition  $P$  composed of  $n$  tuples is given by the ratio  $dp = \frac{\text{number of tuples containing duplicates in } P}{\text{total number of tuples in } P}$ .

The percentage of duplicates inside a partition is a measure of the quality of data in the partition. If this percentage is low (only a few duplicates present in the partition) the quality of data is high and if it is high (a lot of duplicates present) the quality of data is poor.

**Definition 5.** Given partition  $P = \{t_1, \dots, t_n\}$ ,  $\{v_1, \dots, v_n\}$  the set (without repetitions) of join attribute values inside the partition, and the random variable  $f : \text{String}^n \rightarrow \mathcal{R}$ , that assigns the real number  $x_i$  to attribute value  $v_i$ , the frequency skewness of a  $p$  is given by the third standardized moment of the values frequency distribution inside the partition, by using the formula  $\frac{\sum_k x_k^3 \text{freq}(\text{Attr}(t_k))}{\sigma_f^3}$ , where  $\text{freq}(t_i)$  denotes the frequency of value  $\text{Attr}(t_i)$  inside the partition and  $\sigma_f$  is the standard deviation of random variable  $f$ .

The frequency skewness measures the *dispersion* of join attribute values inside the partition. A high skewness value means that the partition is made by a big number of values having low frequency occurrence and only a few ones with a high frequency occurrence. A low skewness value instead means that all tuples in the partition have more or less the same frequency occurrence.

Before introducing the concept of entropy of a partition, we define the subpartition  $P_P = \{p_1, p_2, \dots, p_m\}$  of partition  $P = \{t_1, \dots, t_n\}$ , with  $m \leq n$ , where every element in  $P_P$  is defined as a consecutive group (bag) of lexicographically ordered tuples inside  $P$ . In order to avoid having many groups of average size equals to 2, even in cases when no order exists in the partition, only groups of 1 tuple or groups of 3 or more tuples are considered. For instance, the subpartition of partition  $P = (abc, abcd, dfk, dfg, dfa)$ , is  $P_P = (\{abc, abcd, dfk\}, \{dfg\}, \{dfa\})$ .

**Definition 6.** Given partition  $P = \{t_1, \dots, t_n\}$ , and the subpartition  $P_P = \{p_1, p_2, \dots, p_m\}$  of the partition,  $m \leq n$ , where the normalized length of  $p_i$  is  $\frac{|p_i|}{n}$ , the entropy of the partition is given by  $-\sum_{i=1}^m p_i \log p_i$ .

The entropy of a partition measures the orderliness of tuples inside the partition with respect to the join attribute value. If the partition is perfectly ordered this value will be low,  $-1 \log 1 = 0$ , and if the partition tuples distribution is randomized this value will be high,  $-\sum_{i=1}^m p_i \log p_i = -\sum_{i=1}^m \frac{1}{n} \log \frac{1}{n} = \log n$ , where  $\forall i. p_i = \frac{1}{n}$ .

## 4.3 MAR: An adaptive approach to the problem

As discussed in the previous section the complexity of the symmetric set hash join algorithm tends to be quadratic in its worst case. In the rest of this section a solution for the problem is presented without entering too much in detail.

### 4.3.1 General introduction of the MAR architecture

The computational complexity of the symmetric set hash join algorithm is quadratic in its worst case and as discussed in section 4.1 the usage of the algorithm should be avoided whenever possible. Figure 4.2 better illustrates this technique. The idea is to use the local characteristics of data at the current execution point in order to estimate the characteristics of the data still to be read. The most important of all this characteristics is the local duplicates percentage. If this percentage is high then the symmetric set hash join should be used and no matches will be lost although the efficiency of the method will decrease. If the percentage is low then a normal join algorithm (e.g. symmetric hash join) should be used instead, all possible matches will continue to be returned and the efficiency of the method will be high.

The concept of changing the execution plan in the middle of a query processing, taking into consideration the local characteristics of data, is well known in literature with the term *adaptive query processing* [DIR07]. In many real life situations data statistics or other kind of information available a priori (before the query execution) may not be accurate or even wrong. As a consequence using this information to take decision about which particular query plan to execute could result in a bad choice (suboptimal plan). Adaptive query processing overcomes the problem by ignoring a priori information and incrementally gathering new one over the data, in parallel with the query

execution. The new gathered information is in contrast less complete but more up to date than the old one. Experimental results demonstrate the efficiency of the technique, many times resulting in the choice of an optimal or near-optimal execution plan [DIR07].

As introduced in [KC03] one of the methods to implement an adaptive behavior inside a system is that of using a *MAPE* (Monitor, Analyze, Plan, Execute) architecture (see Figure 6.2). This type of architecture is formed of two main components

1. a managed component,
2. an autonomic manager.

The managed component is the one that carries over the real work to be done, the join algorithm in our case. The autonomic manager on the other hand continually uses a MAPE approach in order to guarantee that the managed one is always giving the best possible effort

**Monitor** the managed component actual performance and behavior in order to obtain valuable statistics and other type of information,

**Analyze** the previously collected statistics and information to detect problems and/or opportunities,

**Plan** a new behavior for the managed component in order to solve current problems or exploit in the best way the new opportunities,

**Execute** the new elaborated plan.

The engineering effort for us will though be defining more in detail all the four phases of the MAPE approach in order to build an efficient adaptive system that solves the duplicates matching problem.

First of all the parameters to monitor during the join execution should be defined. This parameters must not be static (have the same value over time) but continually change and their current value should give important information about the duplicates presence in the data and as a consequence the fact that matches are being missed. At the end, a detailed description of how and at which point of the join execution the monitoring of this variables is done in practice should be given.

Once the values of monitored parameters are collected there is need to further elaborate this information and draw important conclusions for our system such as estimate the probability that matches are being missed or not. A detailed description of how this Analysis component actually works together with the degree of confidence of its conclusions should be though given. In Figure 4.2 defining this component corresponds in defining the guards of the system state diagram.

Normally, adaptive processing systems elaborate a completely new execution plan while the Analysis component decides there is need for a change. Our system on the other hand is a lot simpler in this direction, there are only two different execution plans corresponding to the execution of a normal join or an approximate one. As a consequence there is no need for a complex Planning component. Our solution is that of melting the two phases of Plan and Execute in a single one called *Respond*. This type of architecture would though be called a *MAR* one.

### 4.3.2 Pros and cons

Adaptive query processing by the MAR approach seems to be the best solution to our problem so far. By adaptively switching the type of join algorithm the computational resources would be used at their best and no many matches would be missed. The MAR approach on the other hand is made of separate functional components. This guarantees all the advantages of modular programming facilitating implementation and testing and making the architecture extremely flexible: One component could be replaced by another one with the same interface, but different behaviors. The

architecture is also sufficiently robust, defects in one of the components can be isolated and repaired without needing to make changes to the others.

The approach has also its negative aspects. First of all adaptive query processing doesn't always guarantee the optimal solution, but the optimal estimated one. This means that in some situations wrong decisions could be derived and the system would fail to be in the right state at the right time: The approximate join could be wrongly used instead of the normal one even no matches are actually being lost, or normal join could be used instead of the approximate one even if matches are being lost (see Figure 4.2). The efficiency of the adaptive approach depends on the percentage of right decisions taken in real life scenarios. The challenge is to build the best possible adaptive system, knowing that it is never going to be perfect. In order to build such a system the three distinct components Monitor, Answer and Respond should be properly defined and this is not straightforward. In chapter 6 a complete description of our MAR architecture is given.

## 4.4 Problem scenarios

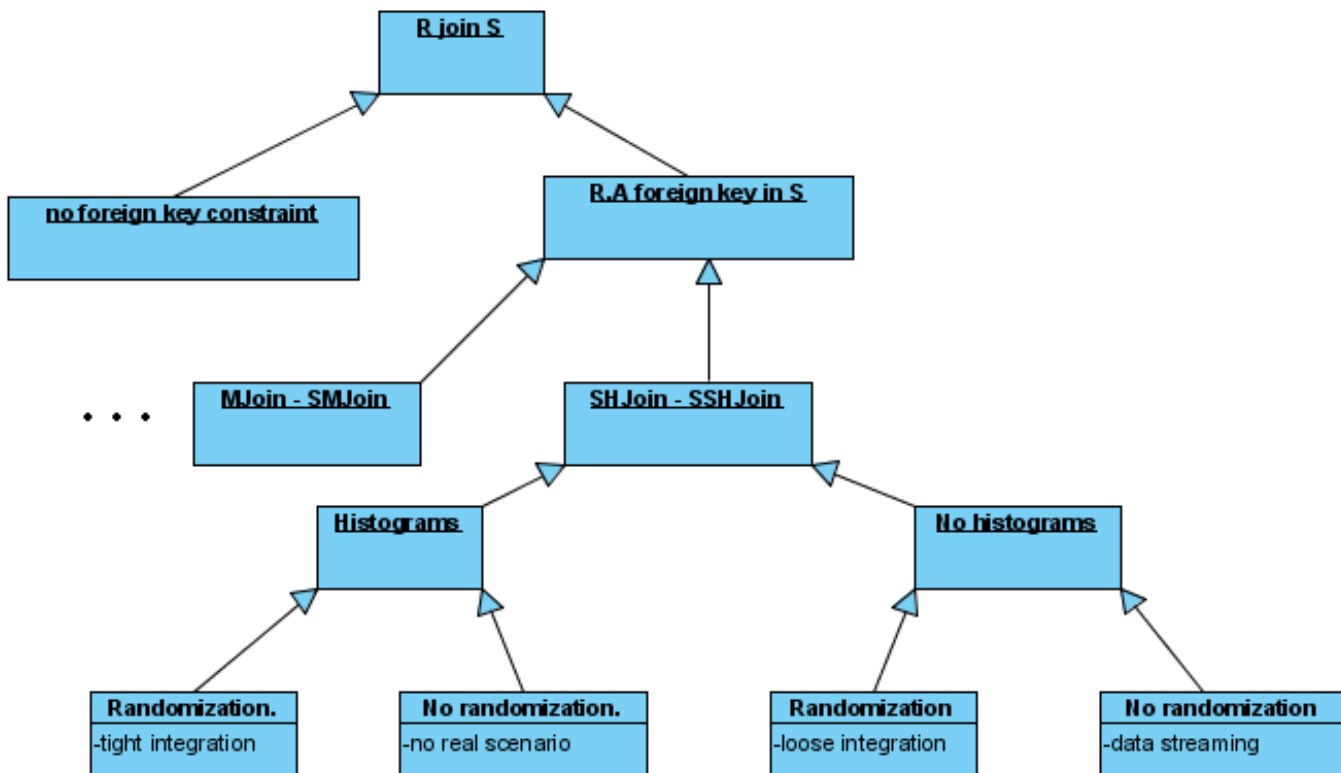


Figure 4.4: Problem representation

In Figure 4.4 all possible scenarios that can happen when coping with duplicate matching at run time are given in a tree diagram form. Every node of the tree represents a possible scenario and every child is a more specific case of the parent node. We are interested in the leaves as these are the more detailed of all scenarios.

The root of the scenarios tree is the more general case, a join between two tables. The nodes at the first level represent generic knowledge we have on the data. In the rest of this thesis we will concentrate on the rightmost scenario of the first level, the one in which one of the tables involved in the join has a foreign key constraint on the other. All nodes at the third level represents couples of specific join algorithms used in the MAR approach

- one normal join algorithm to be used if duplicates are not present

- one symmetric join algorithm to be used if duplicates are present

As mentioned in the previous sections of this chapter join execution will continually switch back and forth between this two types of join depending on local data characteristics. In the figure two examples of couples of algorithms are given

- MJoin–SMJoin
- SHJoin–SSHJoin

The first one corresponds to the merge join algorithm (normal one) and a hypothetical symmetric merge join algorithm (approximate one). The second scenario corresponds to that used in the rest of this thesis where the two involved algorithms are the symmetric hash join (normal) and the symmetric set hash join (approximate) introduced in chapter 3.

Nodes at level four represent the first specialization of the general scenario used in the rest of this thesis depending on additional information we may or may not have on the local frequency distribution of values in the join inputs. This distribution if known is given in the form of histograms, in which for every single value or more generally range of values in the tuples join attribute the frequency occurrence of the value is given. For example if a table contains  $\{a, b, b, b, c\}$  as the only values of the join attribute, its corresponding histogram will contain three entries

1.  $a \rightarrow 1$
2.  $b \rightarrow 3$
3.  $c \rightarrow 1$

The four nodes at level five are leaf nodes and, as previously mentioned, represent the most specific scenarios

**Tight integration** the frequency distribution of the tuples in the table is given in form of histograms. There is also the possibility to randomize the distribution of tuples in the table, obtaining in this way high values of data entropy. This type of scenario represents situations in which both tables are available at the joining site a long time before the join execution begins and there is the opportunity to gather statistics or perform other types of operations (e.g. the joining is performed from the tables owner).

**Loose integration** the frequency distribution of the tuples is unknown, but there is the possibility to randomize the tuples distribution and obtain low data entropy levels. This type of scenario could happen in cases the data provider is different from the one that performs the join operation (data integrator), the table arrived at the integrating site just before the join execution and there is no time for a complete calculation of the frequency distribution. Yet the integrator could simulate a randomized distribution of tuples in the data simply by reading tuples in a random order and not in sequence.

**Data streaming** This is the worst of all scenarios, no information on the data exists, the data is seen only once and then discarded, the results are calculated on the fly. All the possible information over the data would though be gathered during the join execution.

We believe that the second scenario from the left, in which histograms could be calculated, but the distribution could not be randomized, is not possible in practice. In order to obtain a complete frequency histogram of the join attribute values a linear scanning of the data is necessary and during this scanning a randomizing operation could be performed *for free*.

# Chapter 5

## A probability model for the result size estimation

In this chapter the necessity of having a probability model that pilots the decisions made by our adaptive component is reported. Then a series of traditional models are tailored for our application needs and their advantages and disadvantages are discussed together with real-life examples to better understand their usage and limitations. This chapter is organized as follows: Section 5.1 describes the importance of a probability model for our adaptive system. Section 5.2 reports a binomial model tailored for our system needs. Finally 5.3 reports another probabilistic model tailored for the needs of our system, a hypergeometric one, and shows its theoretical advantages with respect to the binomial model.

### 5.1 Why do we need a model for our system?

Our final goal is to build a correct MAR architecture that switches back and forth between normal and approximate join depending on local characteristics of data (see figure 4.2)

1. use normal join if data locally contain no duplicates
2. use approximate join if data contain duplicates locally

The first problem to solve though is to automatically detect the presence of duplicates in the data. This means either that we build an Oracle that classifies tuples either as duplicates or not, or that we use the effects caused by duplicates during the join execution. The only effect we can think of is the same we are trying to avoid: Miss matches between tuples that refer to the same real-life entity but have different representations!

#### 5.1.1 Detection of missed matches

Missing matches means having less tuples in the result size (useful information has been lost). As a consequence, if we knew in advance the exact cardinality of the result size in case all useful information were retrieved, at the end of the join execution we could tell for sure whether matches have been missed or not. In the specific problem we are trying to solve this is straightforward because the foreign key constraint gives an exact estimation of the join result: the result size of the join should be the same of the foreign key's table cardinality. Knowing that duplicates were present only at the end of the join execution may not be sufficient in those scenarios where the join result cannot be recomputed (using an approximate join instead of the exact one, thus avoiding to miss matches). This could be the case of a join between two streams for example, in which the streams are seen only once and the join is computed on the fly. It is important though to have an estimation of the

join result size at every point of the join execution. In this way if matches are being missed and we notice it in time we can still switch to approximate join and stop losing important information. Unfortunately there is no exact formula that gives this quantity, because the symmetric hash join has only a partial vision of the tables until the end of the entire execution. There is the need for a probabilistic model as the ones introduced in section 2.3. The idea is to think at the process of obtaining a certain sequence of result sizes  $rs_1, \dots, rs_M$  during the join execution as a random process which obeys to certain laws. A model that describes such process could then be used to probabilistically detect missed matches.

In Figure 5.1 a symmetric hash join scenario between tables  $R$  and  $S$  is presented. Column  $A$  in table  $R$  is a foreign key for column  $B$ , which is a key in  $S$ . The cardinality of table  $R$  is  $M$ , while the cardinality of table  $S$  is  $N$ ,  $M \geq N$ . Let  $rs_n$  be the observed result size at execution point  $n$ , meaning  $n$  values have been read from both tables. If symmetric hash join between the two tables is executed until its termination the value of the result size at the end of the execution will be one of the following

1.  $rs_M = M$
2.  $rs_M < M$

Since every tuple in table  $R$  can match with exactly one in table  $S$ <sup>1</sup> and the cardinality of the result is equal to the cardinality of  $R$  we are sure there have been no missed matches in the first case. Using the same reasoning we can conclude that in the second case some matches have been lost because of duplicates presence in one of the tables. The number of missed matches is proportional to the number of duplicates.

Suppose now that detecting missed matches only at the end of the join execution is too late. We can make use of a particular probability model that at a generic point  $n$  in the join execution gives an estimation of the result size at this point, denoted by  $E(n)$ . Lets also think of the join execution at point  $n$  as a random experiment with two possible outcomes  $\zeta_1 = \{\text{Some matches are being lost}\}$  and  $\zeta_2 = \{\text{There are no lost matches}\}$ . Comparing the observed result size values with the model estimated ones we can assign a probability to events  $\zeta_1$  and  $\zeta_2$ . Suppose join is interrupted at point  $n$  (Figure 5.1) having  $rs_1, \dots, rs_n$  the sequence of the observed result sizes and  $E(1), \dots, E(n)$  the sequence of the expected result sizes (using the model). From this information we can exactly calculate the probability  $P(\zeta_1)$  of having missed matches. In case this value were too high (e.g.  $P(\zeta_1) = 0.99$ ) we could be enough confident to believe matches are really being missed and immediately switch to approximate join.

### 5.1.2 A model that describes the result size

Let now describe more in detail what a probability model for the join result size is. From definition 2 in 4.2 at the  $n^{\text{th}}$  execution point  $n$  values have been read from both tables  $R$  and  $S$ , all matching tuples to this point have been returned and the join is ready to read a new one from table  $R$ .

**Definition 7.** Let  $J = R \bowtie S$  be a symmetric hash join between tables  $R$  and  $S$ . Let  $T = \{1, \dots, n, \dots, M\}$  be the join execution points. A probability model  $\mathcal{P}$  for the join result size estimation is a family of random variables  $\{X_1, \dots, X_n, \dots, X_M\}$  indexed by values in  $T$ , in which variable  $X_n$  represents the result size value at point  $n$ .

Supposing to interrupt the join execution at a generic point  $n$ , being  $rs_1, \dots, rs_n$  the sequence of the observed result sizes at every execution point, there are at least three possible methods that allow detection of missed matches to this point.

---

<sup>1</sup>column  $B$  in  $S$  is a primary key

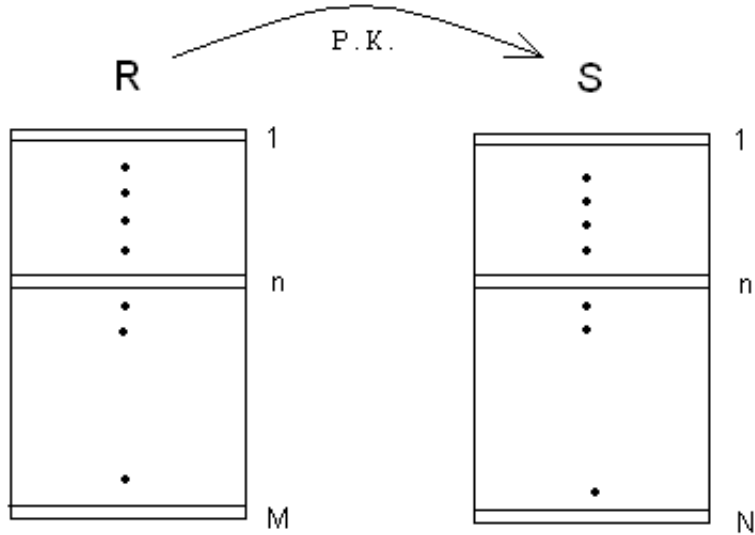


Figure 5.1: A symmetric hash join scenario

### Hypothesis testing

As introduced in section 2.4 given an observed random sample  $(rs_1, \dots, rs_n)$  from a certain distribution  $D(x)$ , hypothesis testing is a powerful tool that probabilistically allows to detect the value of the unknown parameter  $x$  in distribution  $D$ . In order to use hypothesis testing in our system, first of all the assumption *Observed values  $rs_1, \dots, rs_n$  should belong to the same probability distribution* must hold. As we will see in the sections 5.2 and 5.3 this assumption fails to hold because the outcome of every execution point is modeled by a different random variable.

A partial solution could be that of considering a short consecutive sequence of observed result sizes,  $rs_i, \dots, rs_{i+k}$ , with  $k \ll n$ , as belonging to the same distribution. If the sequence is relatively short, that is  $k \rightarrow 0$  then hypothesis testing could be used. The problem in this case will be a low value of the test power since  $k$  is small.

### $n^{th}$ order distribution

The  $n^{th}$  order distribution introduced in section 2.3 is a powerful method that assigns a probability value to the event  $A = \{(rs_1, \dots, rs_n) \text{ are the observed result size values during the join execution if duplicates were absent}\}$ . If this probability is too low then we can be enough confident to conclude that duplicates are present and matches are being missed.

### Chebyshev's inequality

Chebyshev's Inequality states that *No more than  $1/k^2$  of the values are more than  $k$  standard deviations away from the mean.* This can be traduced as

$$P(|x - \mu| \geq k\sigma) \leq \frac{1}{k^2} \quad (5.1)$$

where  $x$  is a possible value in the distributions range.

Returning to our problem of missed matches, given a generic observed result size  $rs_n$  this inequality gives a high bound to the event  $A = \{rs_n \text{ is the observed result size value at point } n \text{ if duplicates}$

were absent}. If such probability is too low we can confidently conclude that duplicates are present and matches are being missed.

## 5.2 A binomial model

In the following the description of a binomial model  $B = \{X_1, \dots, X_n, \dots, X_M\}$  for the join result size estimation is given together with some of its limitations. As we will see, every  $X_n$  in the model is a binomial random variable with parameters depending on  $n$ .

### 5.2.1 Using the binomial model to estimate the result size

A binomial distribution  $B(n, p)$  (see section 2.2) describes experiments in which  $n$  independent Bernoulli trials with parameter  $p$  are performed in sequence. Each Bernoulli trial has two possible outcomes:

1. success with probability  $p$
2. failure with probability  $q = 1 - p$

The event of obtaining a certain result size value  $rs_n$  at execution point  $n$  can be described by a binomial model. Have a look at figure 5.1 again. A symmetric hash join is being executed between tables  $R$  and  $S$  ( $R$  has a foreign key constraint in  $S$ ). Let's stop join execution at generic point  $n$ , after foreign keys  $FK(n) = \{fk_1, \dots, fk_n\}$  have been read from  $R$  and primary keys  $PK(n) = \{pk_1, \dots, pk_n\}$  have been read from  $S$ . From the foreign key constraint the value of  $rs_n$  will be at most  $n$  and directly proportional to the cardinality of  $FK \cap PK$ . The two extreme cases will be

- $rs_n = n$  if  $FK = PK$
- $rs_n = 0$  if  $FK \cap PK = \emptyset$

In the general case a tuple  $t$  from  $R$  will have a match in  $S$  if  $Attr(t) = v \in PK$  (key  $v$  has been read from the primary table before). Remember that function  $Attr$  takes in input a tuple from the join inputs and returns the join attribute value of the tuple as output. If  $t$  doesn't match any value in  $S$  one of the following should be true

1. The  $v$  key from table  $S$  has not been read yet, because it belongs to the still to be read portion of  $S$  (the key will be read in the future, before the algorithm ends and the match will be returned as an element of the result)
2.  $v$  is a duplicate and the corresponding key  $v_1$  in  $S$  doesn't exactly match  $v$  ( $v_1$  has been read in the past or should still be read, a match would be missed anyway)

The following example illustrates the idea over two concrete tables.

**Example 15.** *Suppose a join between table Orders and Clients (Figure 5.2), the first having a foreign key constraint on the second, has been performed till execution point 3.  $FK = \{\text{Bill Gotes, Roald Lengu, Steve Jobs}\}$  and  $PK = \{\text{Roald Lengu, Bill Gates, John Smith}\}$ . There is no match for foreign key Bill Gotes because it is a duplicate, there is a match for Roald Lengu because the key with the same value is read from Clients and there is no match for Steve Jobs because such key is not yet read. If we continue the execution of the join till completion foreign key Steve Jobs will find a match, but Bill Gotes will not.*

Client	Item	Quantity
Bill Gotes	Windows Millenium	1
Roald Lengu	Prosciuto Crudo	3
Steve Jobs	IPod	1
John Smith	Iron	3

(a) Orders

Client	Age	Address
Roald Lengu	24	Via Camogli
Bill Gates	55	Piazza Microsoft
John Smith	30	Notting Hill
Steve Jobs	50	Corso Apple

(b) Clients

Figure 5.2: A join scenario between *Orders* and *Clients* tables

Lets now describe more in detail a binomial model for the result size estimation at point  $n$ . A success is defined as the event  $S = \{fk \in FK \wedge fk \in PK\}$  (a match is found), while a failure as the event  $F = \{fk \in FK \wedge fk \notin PK\}$  (match not found). Supposing there are no duplicates present and the probability of reading a generic foreign key in every execution point is the same for all foreing keys, the probability of success at point  $n$  is given by the *number of keys read, total number of keys ratio*

$$p = n/N \quad (5.2)$$

where  $N$  is the cardinality of table  $S$ .

The assumption that all foreign key values are equiprobable is used to obtain indipendence of single events (single match or single missed match). Without this assumption the binomial process is invalid. Assumption stating that duplicates are not present is the one we want to test by using one of the following methods

1.  $n^{th}$  order distribution
2. Chebyshev's inequality

Let  $X_n$  be the binomial variable  $B(n, n/N)$  that counts the number of success to point  $n$  with  $p = n/N$  and  $q = 1 - n/N = (N - n)/N$ . The mean and variance of  $X$  will be

$$\mu_X = \frac{n^2}{N} \quad (5.3)$$

$$\sigma^2_X = \frac{n^2(N - n)}{N^2} \quad (5.4)$$

In the first case the cumulative distribution functions  $F_X = P(X_n \leq rs_n)$  is used

$$F_X = \sum_{i=0}^{rs_n} \binom{n}{i} p^i (1-p)^{n-i} \quad (5.5)$$

and in the second the inequality

$$P(|rs_n - n^2/N| \geq kn^2(N - n)/N^2) \leq \frac{1}{k^2} \quad (5.6)$$

Hypothesis testing method cannot be used with this model because as announced in 5.1.2, the sequence of observed result sizes  $rs_1, \dots, rs_n$  does not belong to the same distribution. Indeed the  $i^{th}$  value in this sequence ( $rs_i$ ) is the outcome of binomial random variable  $X_i$  (belongs to this particular distribution), which is different from all other random variables  $X_j, j \neq i$ .

In Example 16 the usage of this methods in a real case scenario is illustrated.

**Example 16.** Consider again the join in example 15 in which execution is stopped at point  $n = 3$  and the result size value is  $rs_n = 1$ . The value of  $p$  can be estimated as  $n/N = 3/4$ , the expected value  $E(n) = 2.25$  from Equation 5.3 and the variance as  $\sigma^2 = 0.75$  from Equation 5.4 ( $\sigma \approx 0.87$ ).

By using the  $n^{\text{th}}$  order distribution method

$$P(X \leq 1) = F_X(1) = \sum_{i=0}^1 \binom{3}{i} \left(\frac{3}{4}\right)^i \left(\frac{1}{4}\right)^{3-i}$$

$$P(X \leq 1) = \binom{3}{0} \left(\frac{3}{4}\right)^0 \left(\frac{1}{4}\right)^3 + \binom{3}{1} \left(\frac{3}{4}\right)^1 \left(\frac{1}{4}\right)^2$$

$$P(X \leq 1) = \left(\frac{1}{64}\right) + \left(\frac{9}{64}\right)$$

$$P(X \leq 1) = \frac{10}{64} \approx 0.15$$

As this probability is relatively low, we can conclude that matches are being missed (which is true in example 15). If the value of  $rs_3$  was 2 instead of being 1, by using the same formula we could conclude that  $P(X \leq 2) = 37/64 \approx 0.57$ . In this case we should conclude that most probably no matches are being missed and should not worry about losing important information.

By using Chebishev's inequality

$$P(|1 - 2.25| \geq 0.87k) \leq \frac{1}{k^2}$$

$$P(1.25 \geq 0.87k) \leq \frac{1}{k^2}$$

$$1.25 \geq 0.75k \Rightarrow k \leq 1.44 \Rightarrow 1/k^2 \geq 0.48$$

$$P(|rs_n - 2.25| \geq 1.2528) \leq 0.48$$

In this case we cannot be much confident that duplicates are being missed.

Notice that Chebishev's inequality gives only an upper bound of the probability we want to estimate, while  $n^{\text{th}}$  order distribution exactly calculates it. By using the second method our level of confidence is lower than by using the first. On the other hand, if the value of  $n$  was big enough using the second method would be less expensive than using the first which would require the computation of a large sum of binomial coefficients.

## 5.2.2 Limitations of the binomial model

As stated in *modern heuristics* ([MF04]) a model is a simplification of the real world problem, otherwise it would be as complex and unwieldy as the natural setting itself. Once the model has been created a solution can be found. It is important to understand though that the solution is only a solution in terms of the model. If the model has a high degree of fidelity, we can have more confidence that our solution can be meaningful. On the other hand if the model makes too many simplifications and unreal assumptions the solution may be meaningless or coarse.

Lets now see more in detail the assumptions made by the previously introduced binomial model and discuss their fulfillment degree. The only assumption we made was that the probability of reading a certain foreign key  $fk$  at the  $n^{\text{th}}$  execution point is the same for any key and for any point. This assumption fully holds only in if the following statements are true

1. Foreign key table has infinite values
2. The entropy of this table is high (value ordering is low, the table is randomized)
3. Values are uniformly distributed

See Example 17 for e scenario in which unsatisfaction of these statements causes unfulfillment of the assumption.

**Example 17.** Consider again the tables in example 15. We can see that table Orders has finite dimensions and every tuple in the table is presented with frequency  $f = 1$ . The probability of reading value Bill Gotes before beginning to execute the join at any execution point is

$$p = \frac{\text{Bill Gotes still to be read}}{\text{Values still to be read}} = \frac{1}{4}$$

After reading the first Bill Gotes value, the probability of reading again the same value in the rest of the table would be

$$p = \frac{\text{Bill Gotes still to be read}}{\text{Values still to be read}} = \frac{0}{3} = 0$$

As we see the probability of finding value Bill Gotes is different in different execution points, while the binomial model assumes single success probabilities doesn't depend on time. Suppose now table Orders was infinite. If values in the table were ordered, then after reading value Bill Gotes the probability of reading again the same value or a lexicographically close one would be higher than reading a generic value. If table Orders was infinite and randomized but values were not uniformly distributed, e.g. Bill Gotes had more occurrences than others during a certain execution period (a lot of orders were made by this person at a certain date), the probability of reading this value instead of a generic one would be higher again.

In real life scenarios the first statement is never satisfied either because there are no infinite stored tables or because joins between infinite streams are only executed considering finite windows over such streams[cite a paper...]. The statement can only be relaxed for values of  $n$  a lot smaller than the real tables cardinality (considering the remaining tuples to be read as nearly infinite).

The second statement is somewhat more probable in real life situations. First of all the probability that values in a given stream are presented in perfect order is low (1/cardinality of the table), unless there is a particular reason. Just to mention one possible reason think about a stored table in which tuples are ordered according to a certain attribute to facilitate the use of merge join. Second, in many cases even if values are ordered, there is the possibility to randomize their distribution before performing the join. In others, such as in stream join scenario, total randomization is impossible because of time or space constraints. One solution here would be that of using a finite dimension window to perform partial randomization (over the tuples of the window).

The third statement depends on the table's domain and nothing can be done by the binomial model to improve things. In a names database for example, some names like *John* or *Bill* will be more frequent than others like *Roald*.

## 5.3 A hypergeometric model

In the following the description of a hypergeometric model  $B = \{X_1, \dots, X_n, \dots, X_M\}$  for the join result size estimation is given. Every  $X_n$  in the model is a hypergeometric random variable with parameters depending on  $n$ . At the end of this section some limitations of the model are discussed and a confront with the binomial one is made.

### 5.3.1 Using the hypergeometric model to estimate the result size

A hypergeometric distribution  $H(N, k, n)$  (see section 2.2) describes experiments in which  $n$  draws without replacement are made from a population of size  $N$ , having  $k$  defective objects. The corresponding random variable counts the number of defectives in the drawn sample.

The event of obtaining a certain result size value  $rs_n$  at execution point  $n$  can be described by a hypergeometric model. Have a look at figure 5.1 again. A symmetric hash join is being executed between tables  $R$  and  $S$  ( $R$  has a foreign key constraint in  $S$ ). Let  $M$  be the cardinality of table  $R$  and  $N$  that of table  $S$ . Suppose join execution is stopped at generic point  $n$ , being  $FK = FK(n)$  and  $PK = PK(n)$  the set of foreign and primary keys read to this point respectively.

Similarly to the definition of a success in 5.2 a defective can be defined as a foreign key  $fk \in FK$  that also belongs to  $PK$ . Intuitively a defective is a value from the foreign key table whose analogue primary key value has already been read from the other table (corresponding in a match). A defective though corresponds to a match and the number of defectives read from the foreign table to the cardinality of the result size at a certain join execution point.

A hypergeometric process that describes the result size estimation is a family of random variables  $\{X_1, \dots, X_n, \dots, X_M\}$ , indexed by values in the join execution points  $1, \dots, n, \dots, M$ , in which generic variable  $X_n$  represents the result size at point  $n$ . The mean and variance of this random variable are

$$\mu_X = \frac{nk}{N} \tag{5.7}$$

$$\sigma_X^2 = \frac{n \frac{k}{N} \frac{N-k}{N} (N-n)}{N-1} \tag{5.8}$$

$k$  represents the number of defectives in the *whole table*  $R$  (as a bag). If values frequency distribution in  $R$  is known, because histograms are used, and table's entropy is high (it is not ordered)  $k$  can exactly be calculated by formula

$$k = \sum_{fk \in PK} H(fk), \quad \text{where } H(fk) \text{ is frequency of } fk \text{ in table } R \text{ given by histogram } H \tag{5.9}$$

If the exact distribution is unknown, but the frequency distribution of  $R$  is not highly skewed (all values have almost the same occurrence frequency) and the entropy is high the following formula could be used to calculate  $k$

$$k \approx \frac{nM}{N} \tag{5.10}$$

High entropy values guarantee a certain degree of independence between single random variables in the process. The idea behind equation 5.10 is simple. As frequency occurrence of values in the foreign table, of cardinality  $M$ , is uniform (same for all) and there are exactly  $N$  distinct join attribute values present in  $R$  (equal to the number of keys in  $S$ ), each value will occur with frequency  $M/N$ . If  $n$  primary keys have already been read from  $S$ , the number of defectives in  $R$  will be  $nM/N$ ,  $n$  times more than for one single value.

Assuming that the values frequency distribution in the foreign table is known (or not highly skewed) and that the entropy of the same table is high the presence of duplicates in the tables can be tested by one of the following methods

1.  $n^{th}$  order distribution
2. Chebyshev's inequality

In the first case the cumulative distribution function  $F_X(x) = (X \leq x)$  can be used

$$F_X(x) = \sum_{x=0}^n \frac{\binom{k}{x} \binom{N-k}{n-x}}{\binom{N}{n}} \tag{5.11}$$

and in the second Chebyshev's inequality

$$P(|rs_n - nk/N| \geq \alpha\sigma) \leq \frac{1}{\alpha^2} \tag{5.12}$$

where  $\sigma = \sqrt{\frac{n \frac{k}{N} \frac{N-k}{N} (N-n)}{N-1}}$  is the standard deviation and  $\mu = nk/M$  in the mean of random variable  $X$ .

As in the binomial model, hypothesis testing technique cannot be used here either. The assumption that all observed size values belong to the same distribution does not hold. Indeed, two different result sizes  $rs_n$  and  $rs_m$  observed at different execution points  $n$  and  $m$  ( $n \neq m$ ) respectively are represented as the outcomes of hypergeometric variables  $X_n = H(N, n, k_1)$  and  $X_m = H(N, m, k_2)$

in our model. As we can see, these variables have at least one different parameter, the second one. The third parameter  $k_1$ , for the first variable, and  $k_2$  for the second can be calculated by one of the methods previously described in this section. Example 18 illustrates the application of both methods to a real scenario, showing the superiority of the hypergeometric one.

**Example 18.** Consider again the join between tables in example 15 stopped at execution point  $n = 3$ , being the result size at this point  $rs_3 = 1$ . The expected value can be estimated as  $\mu_X = 2.25$  by Equation 5.7 and the variance as  $\sigma^2 = 3/16$  ( $\sigma \approx 0.43$ ) by equation 5.8. Using equation 5.9 the value of  $k$  corresponds to the values in the foreign key table that have already been read from the primary key one. If value Bill Gotes was not a duplicate this values should be equal to 3. Using equation 5.10 the value of  $k$  is 3 again.

According to  $n^{\text{th}}$  order distribution method

$$P(X \leq 1) = F_X(1) = \sum_{x=0}^1 \frac{\binom{3}{x} \binom{1}{3-x}}{4}$$

$$P(X \leq 1) = \frac{\binom{3}{0} \binom{1}{3}}{4} + \frac{\binom{3}{1} \binom{1}{2}}{4}$$

$$P(X \leq 1) = 0 \text{ Remember that } \binom{n}{k} \text{ is 0 if } k \geq n.$$

As this probability is 0 we could conclude that matches are being missed and execution should switch to approximate join. If the observed value at this point was 2 instead of 1, using the same formula we could calculate  $P(X \leq 2) = 0.75$ . In this case we should continue with the normal join execution and suppose matches are not being missed. Using Chebyshev's inequality

$$P(|1 - 2.25| \geq 0.43\alpha) \leq 1/\alpha^2$$

$$(1.25 \geq 0.43\alpha) \leq 1/\alpha^2$$

$$1.15 \geq 0.43\alpha \Rightarrow \alpha \leq 0.43 \Rightarrow 1/\alpha^2 \geq 0.12$$

$P(|rs_n - 2.25| \geq 1.25) \leq 0.12$  As a conclusion even with this method, that is a coarser one, we could conclude that most probably matches are being missed. If the observed result size value was 2 instead of 1 using the same formula  $P(|rs_n - 2.25| \geq 0.25) \leq 1$ .

The important things to notice here are the differences with the binomial model. Using the  $n^{\text{th}}$  order distribution the probability calculated by the binomial model was coarser (0.15 instead of 0) resulting in a lower level of confidence in the assumption that matches were being missed. The same conclusion could be derived even for Chebyshev's inequality method. The upper bound calculated by the binomial model in this case was so coarse (0.48) that the returned result suggested no matches were being missed, while the hypergeometric model (0.12) rightly suggested the opposite.

### 5.3.2 Limitations of the hypergeometric model

As ilustated in Example 18 the hypergeometric model overcomes some of the limitations presented by the binomial one by relaxing some of its assumptions. According to *modern heuristics* ([MF04]), the first model is a better approximation to the real world problem we are trying to solve resulting in a higher level of significance for its suggested solutions. The assumptions made by the binomial model were

1. Foreign key table has infinite values
2. The entropy of this table is high
3. Values are uniformly distributed

The first and the third assumption are not required by the hypergeometric model.

In the binomial case foreign table had to be infinite in order to guarantee independence of single matches (Bernoulli experiments). The hypergeometric model not only doesn't require it, but rightly assumes the opposite, that single matches depend on previous ones (see Example 17).

The third assumption is not required only in case the distribution of values in the foreign table is known (e.g. by histograms). A known distribution allows to calculate the number of defectives  $k$  at every execution point, which is also the only *difficult* parameter of the model to be calculated.

The assumption that levels of entropy should be high must still hold for the hypergeometric model. To understand the reason lets try to get back to the most ordinary situation described by this model: A sample of  $n$  objects is drawn from a population of  $N$ ,  $k$  of which are defective. If the draws were not made randomly, but from a partition of the population made only by defectives (see Figure 5.3), all elements in the sample would be defectives and the hypergeometric model results would not be significant. The same idea can be applied to the symmetric hash join scenario. If foreign key table is ordered than during the execution join long horizontal partitions made only by values already read from the primary key table (defectives), or others from values still to be read (not defectives) could be met.

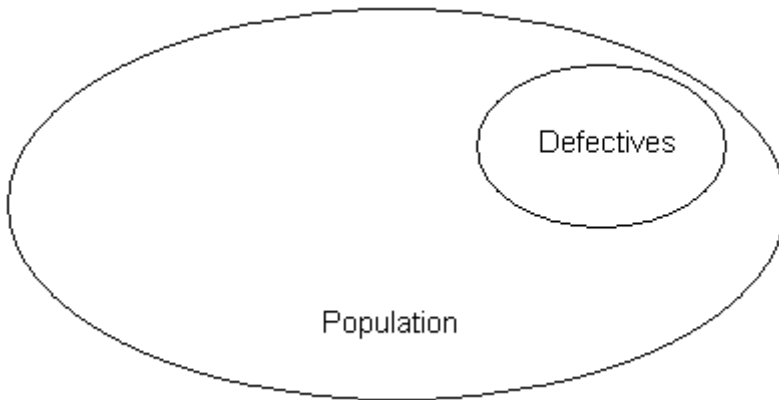


Figure 5.3: A bad sampling example

The only assumptions hypergeometric model makes are the following

1. The entropy of the foreign table is high
2. There is a way to estimate  $k$  at every execution point

To understand how the assumption over high levels of entropy could be overcome see limitations of the binomial model in section 5.2.

The values of  $k$  (number of defectives-matches) can be calculated by one of the equations 5.9 or 5.10 introduced in the previous subsection. If the first equation is used the value of  $k$  is exactly calculated and the model results have a high degree of confidence. The second one gives an approximation for the value of  $k$  assuming that values in the foreign table have approximately the same occurrence frequency. The model's degree of confidence in this case will be directly proportional to the validity of the assumption. The more uniform the frequency distribution is, the more confidence should be given to the model; the more skewed the distribution the less confidence should be given to the model.

# Chapter 6

## A MAR approach to result completeness

In this chapter the MAR approach for the adaptive component of our system, introduced in section 4.3 is instantiated and described in detail. This component is responsible for a fast and appropriate response from the system to changes happened in its dynamic environment, i.e. the missing or not of matches in the join result size and as a result the use of an approximate or exact algorithm respectively to perform an equi-join operation. The rest of this chapter is organized as follows: In section 6.1 the overall MAR architecture is described as inspired by previous work in adaptive query processing. In section 6.2 the monitoring component is described in detail. Section 6.3 gives a detailed description of the second MAR component, namely analyze. Finally, section 6.4 gives the details of the last MAR component, namely respond.

### 6.1 Introduction

*Systems manage themselves according to an administrator's goals. New components integrate as effortlessly as a new cell establishes itself in the human body. These ideas are not science fiction, but elements of the grand challenge to create self-managing computing systems.* This is how the IBM paper on *Autonomic Computing* begins [KC03]. In the following it warns of an imminent software complexity crisis, which is going to deteriorate in the next years. Systems are becoming even more complex, difficult to install, configure, tune and maintain. A great effort is required by engineers and systems administrators to maintain systems operative and in the top of their possibilities. In the future the complexity of these systems will reach certain levels that it will be almost impossible even for the most skilled administrators to deal with everyday problems.

The only option remaining is *autonomic computing* or *adaptive systems*, computing systems that can manage themselves, given high-level objectives from administrators [KC03]. To notice that the *autonomic* term has a biological connotation. In the same way the nervous and hormonal systems govern low level but still several vital aspects of our body, such as the temperature and heart beat, leaving our conscious mind out of this, the autonomic systems will govern themselves without having to involve human beings. Another example of autonomic government is that of the free markets in which the laws of demand and offer maintain prices in specific levels. Thus, it is believed to be profitable seek inspiration in the self-governance of social and economic systems as well as purely biological ones [KC03].

IBM frequently cites four aspects of self-management, which Figure 6.1 summarizes [KC03], making a comparison with current-computing architectures. According to the *self-configuration* aspect system administrators should only give high level objectives to their systems, while all the low level configuration work is going to be carried out automatically from the system. The *self-optimization* aspect will force the systems to continually optimize their performance and efficiency, being always at the top of their possibilities, while current systems require the constant supervising of administrators to continually and manually tune specific parameters. As we will see in the remaining

Table 1. Four aspects of self-management as they are now and would be with autonomic computing.		
Concept	Current computing	Autonomic computing
Self-configuration	Corporate data centers have multiple vendors and platforms. Installing, configuring, and integrating systems is time consuming and error prone.	Automated configuration of components and systems follows high-level policies. Rest of system adjusts automatically and seamlessly.
Self-optimization	Systems have hundreds of manually set, nonlinear tuning parameters, and their number increases with each release.	Components and systems continually seek opportunities to improve their own performance and efficiency.
Self-healing	Problem determination in large, complex systems can take a team of programmers weeks.	System automatically detects, diagnoses, and repairs localized software and hardware problems.
Self-protection	Detection of and recovery from attacks and cascading failures is manual.	System automatically defends against malicious attacks or cascading failures. It uses early warning to anticipate and prevent systemwide failures.

Figure 6.1: Four self-management aspects

of this chapter, we are mainly concerned with this aspect of autonomic computing. According to the *self-healing* concept, systems will autonomically diagnose and solve newly raised problems in the best way possible. Finally, *Self-protection* will allow systems to defend against malicious users or other communicating systems.

Early autonomic systems may treat these aspects as distinct, with different product teams creating solutions that address each one separately. As previously mentioned, in this thesis we only deal with the self-optimization technique. Complex middleware, such as *WebSphere* [web], or database systems, such as *Oracle* or *DB2*, may have hundreds of tunable parameters that must be set correctly for the system to perform optimally, yet few people know how to tune them, resulting in a mandatory autonomic solution.

In this thesis we try to automatically choose the best join algorithm to be used according to local data characteristics

- shjoin, if locally data have no duplicates,
- sshjoin, if there are duplicates locally present in the data.

In the first case, we use a more efficient algorithm still retrieving high quality results. In the second, we are forced to use a less efficient algorithm and still guarantee high quality levels for the results. The challenge is to build a system that automatically choses the right configuration( i.e. join algorithm) in the right moment.

As Figure 6.2 reports, an autonomic element will typically consist of one or more *managed elements* coupled with a single *autonomic manager* that controls and represents them [KC03]. A managed element is an entity that performs the job required by the system. It may be a processor, a query evaluation engine, a web service etc. These components are the ones that already exist in current architectures. The autonomic manager, on the other hand, distinguishes the autonomic element from its non-autonomic counterpart. By monitoring the managed element and its external environment, and constructing and executing plans based on an analysis of this information, the autonomic manager will relieve humans of the responsibility of directly managing the managed element.

In Figure 6.2 the autonomic manager is divided in 4 functional units each carrying out a specific task. The idea is that though an infinite loop over this tasks the performance of the managed element is always maintained to its best, even in an ever-changing execution environment

**Monitor** the managed component actual performance and behavior in order to obtain valuable statistics and other type of information,

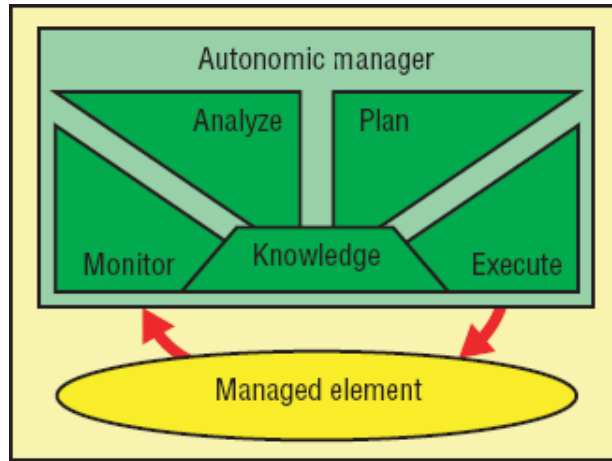


Figure 6.2: The MAPE architecture

**Analyze** the previously collected statistics and information to detect problems and/or opportunities,

**Plan** a new behavior for the managed component in order to solve current problems or exploit in the best way the new opportunities,

**Execute** the new elaborated plan.

From the idea of the MAPE architecture we built a new one called *MAR*, *monitor, analyze, respond*, melting the *plan* and *analyze* phases in a single *respond* one. This was done because in our system there is no need for a complex *plan* phase, as there are only two executable plans depending on the type of the join algorithm to be executed: the normal shjoin or the approximate sshjoin.

In the remainder of this chapter we will see in more detail how each of the 3 previously introduced phases is implemented and works in our adaptive system.

## 6.2 Monitor

As suggested by figure 4.2 there are two basic scenarios we should concentrate in the specification of a MAR architecture.

In the first scenario, the normal shjoin algorithm is being executed, and some parameters should be monitored over time. The series obtained as a result should further be analyzed in order to decide if it is appropriate to continue with the exact join execution, or it is better to switch to the approximate one.

In the second scenario, the approximate sshjoin is being executed and some parameters should though be monitored over time and the series obtained should then be analyzed in order to decide if a switch to the exact join is necessary or it is better to continue with the approximate join execution.

In the first case, normal join is being executed, section 5.1 suggests that the join result size value together with the number of tuples read from each input operator should be monitored over time. In case the first of these values is too low according to the estimation calculated by a certain model than a switch is necessary. Figure 6.3 gives the idea of how these variables are monitored in practice.

The procedure is quite simple. Supposing an exact join is being executed between operators  $R$  and  $S$  that follow the *iterator* approach introduced in section 3.1, the current result size could be obtained iteratively in the following way

- At the beginning the result size value is 0, no result tuple has been returned yet.
- Every time a result tuple is returned the result size value is incremented by one.

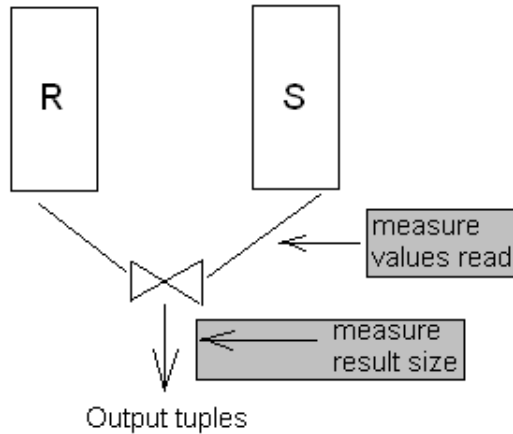


Figure 6.3: Result size monitoring

A similar approach could be also used for computing of the number of tuples read from each input operator

- at the join beginning this value is 0,
- every time the *next* method is called by one of the input operators the value corresponding to that operator is incremented by one.

In the second case, that is, if an approximate join is being executed, the problem is a little more intricate. We must be able to understand that the approximate join actually is not so necessary, that we could use a more efficient exact join instead and still return most, or almost all, of the result. This is the case in which previously duplicates have been present in the input operators and an approximate matching method was necessary to avoid losing *semantically* correct matches. But currently there are no duplicates being read from the join inputs, and an exact matching method is enough to return all *semantically* correct results.

One solution could be that of monitoring the standardized similarity value between valid approximate matches over time. Given two tuples  $t_1$  and  $t_2$  the standardized similarity value between their respective join attributes varies between 0, meaning that the attributes are completely different, and 1, meaning the attributes are identical and the match is exact. A fixed size window could be then used in the algorithm in figure 3.4 to maintain the last  $k$  approximate matches similarity values, for some  $k$ . If the percentage of 1 in this window is sufficiently high, e.g. nearly 100%, then the approximate join algorithm is not necessary and the exact one could be used instead.

In our *sshjoin* algorithm this monitoring could easily be performed by first declaring a boolean array of size  $k$ , which values correspond to the last  $k$  matching similarity results in the following way

- *false*, if the corresponding similarity is less than 1, not an exact match,
- *true*, if the corresponding similarity is equal to 1, an exact match.

The values of the array can be updated in order every time a valid match is returned in line 26, using the value of the *counter* field standardized by the join attributes length as the similarity value (see algorithm in figure 3.4). The total number of *true* values contained in the array could then be used to count the percentage of exact matches in the last  $k$  approximate ones. Such a percentage value, namely *perc*, can be computed incrementally without compromising the overall performance

of the algorithm, maintaining the current value in a local variable and incrementing or decrementing it by a constant cost every time a new match is returned.

In the following section a description of how the series of monitored values can be further used to take decisions over the correct type of algorithm to use according to the local data characteristics, is given.

## 6.3 Analyze

Providing the parameters which values should be monitored over time and the way this process is done in practice is only the first step of building a correct MAR architecture. The sequence of monitored values should subsequently be analyzed in order to make important decisions and as a consequence maintain the system to the top of its computational and quality performance.

### 6.3.1 Exact-approximate join switch

In this scenario, the system is currently executing the normal join between the input operators and continually monitoring the values of the result size at every execution point, see section 4.2, as described in the previous section. If we though at the value of this variables as describing an internal state of the system, then a state transition diagram such as the one in figure 6.4 could describe the behavior of the system over time.



Figure 6.4: Test over exact-approximate switch

At every point, or more generally every  $k$  points, an analysis of such values could tell if the system should continue with the exact join execution or switch to the approximate one, because results are being missed. As introduced in section 5.1, a probabilistic model provides us with a result size estimation at every execution point as well as the probability that a certain sequence of observed result sizes could occur in practice if no duplicates were present.

Section 5.2 suggests the usage of a binomial model for the result size estimation, while section 5.3 suggests a hypergeometric one showing the advantages of the latter over the former. One of the good characteristics of our MAR architecture is that it does not matter which probabilistic model is being used as long as it provides the required estimation and probabilities. This means that not only one of the previous methods could be used, but others could be added as well in the future, each with its advantages and disadvantages. As a consequence, the one with the best characteristics could be used from the beginning to the end of the join execution or the adaptive approach could also be spread to the choice of the right model at the right moment.

Supposing to use a specific model from the beginning to the end of the join execution, which is also the approach followed in the current implementation, as figure 6.4 reports, at execution point  $p$  a decision has to be made

- switch to approximate join if the probability of duplicate occurrences computed by the model is high enough,
- continue with normal join if such probability is low.

A reason to adaptively change the probabilistic model during the join execution could be that of achieving an acceptable trade-off between efficiency and effectiveness. Suppose that execution starts by using a certain computationally expensive but accurate model  $M_1$ . If after having executed part of the join it results clear that the method is not strictly needed to comply with quality constraints a less accurate and less computationally expensive model could be used instead.

For more detailed examples of the different types of models used and the way probabilities are calculated see examples 16 and 18. In the first example the binomial model usage is illustrated and in the second the hypergeometric one.

### 6.3.2 Approximate-exact join switch

In this scenario an approximate join is being executed between the input operators and the value of *perc*, the percentage of times an approximate match is also an exact match, is being monitored over time as described in the previous section. If we think at the value of *perc* as an internal state of the system, then figure 6.5 could describe the system state diagram after  $p$  approximate matches have been returned.



Figure 6.5: Test over approximate-exact switch

Intuitively an exact match is a particular case of an approximate match, in which the standardized similarity between the two tuples join attributes is 1. This means that an approximate join will return at least all the results of an exact one. In addition an approximate join returns also approximate matches, due to the presence of duplicates, that an exact join would certainly miss.

If during a certain execution period, of the approximate join, all, or almost all, approximate matches correspond to exact ones, meaning  $perc \rightarrow 100\%$ , there is no reason for continuing with the approximate join execution. A more efficient exact join algorithm could be used instead and results would still continue to be returned, avoiding to have (too many) miss matches. If on the other hand the value of *perc* is low, then there is still need for the approximate join to be executed in order to avoid missing many matches.

Example 19 illustrates in a real scenario how the method works.

**Example 19.** *Suppose an approximate join is being executed between input operators  $R$  and  $S$ . Let the similarity threshold be 0.75, meaning that tuples whose join attributes have a standardized similarity greater-equal to 0.75 are expected as valid matches. The similarity values between the matching tuples join attributes of the last 5 matches are presented in figure 6.6. Match number 1 refers to the oldest of the five latest matches and match number 5 to the last of all. Suppose also that the lowest value of *perc* for the analysis component to make a switch decision is 70%.*

Match#	Similarity value
1	0.8
2	1
3	1
4	1
5	1

Figure 6.6: Last 5 *perc* values

By calculating the average value of *perc* in the last five matches it comes out that this value is 80%. As it is bigger than 70%, the lowest limit, a switch decision is drawn and execution continues with normal join.

If on the other hand the similarity value of match number 2 was 0.9, instead of 1 than the average value of *perc* over time would be 60%, meaning lower than 70%, the lowest limit. In this case a negative switch decision would be drawn by the analysis component and execution would continue with approximate join.

## 6.4 Respond

Having introduced the monitoring and analysis components of our MAR architecture let's now see more in detail *respond*, the last one. During this phase, according to the decision previously taken by the analysis component the right action is taken

1. continue with the same type of algorithm used so far,
2. switch to the other type of algorithm.

Both cases correspond to two different scenarios. In the first, execution continues either with the exact shjoin or with the approximate sshjoin. In the second, if an exact shjoin was being executed, we switch to the approximate sshjoin one and if approximate sshjoin was being executed switch to the exact shjoin one. As we can imagine the first two cases are not interesting as execution continues in the same way and no changes are made by the architecture. In the second, as an algorithm switch is made, there is need for some of the new algorithm data structures to be initialized taking in consideration the values contained in the old algorithm data structures. The only interesting data structures to mention here are the different hash tables used by the different types of algorithms for their input operators (one hash table for each operator, as both algorithms are symmetric)

- attribute based hash tables used by shjoin,
- qgram based hash tables used by sshjoin.

Suppose shjoin has been executed for a period of time during which  $n$  tuples have been read from both input operators. As a consequence there will be  $n$  tuples in each of the operators hash tables, each tuple stored in the bucket corresponding to the hash value of its join attribute. All the probing operations performed during the algorithm use the value of the join attribute. Suppose the analysis component decides that a change to sshjoin should be made. The hash tables based on join attributes are not useful any more, as the probing phase now is done on qgram values. There is thus the need to obtain all  $n$  tuples maintained in the old attribute based hash tables and *put* them in some new built qgram based ones. A same reasoning can be made for the shjoin-sshjoin switch, in this case transforming a qgram based hash table in an attribute based one. Figure 6.7 reports the hash tables type transformation. As we can see the qgram hash tables are represented bigger than the attribute ones as they normally need more space (see section 3.4.2 for a spatial complexity analysis of sshjoin).

We think it could be helpful to have at least an idea of the computational complexity of the hash tables transformation and see how it impacts on the overall performance of the managed component. Supposing  $n$  values have been read from both input operators and that the average length of the join attribute is  $|jattr|$  the cost of the transformation is in

- $O(|jattr| * n)$  in the case of shjoin-sshjoin
- $O(n)$  in the case of the sshjoin-shjoin

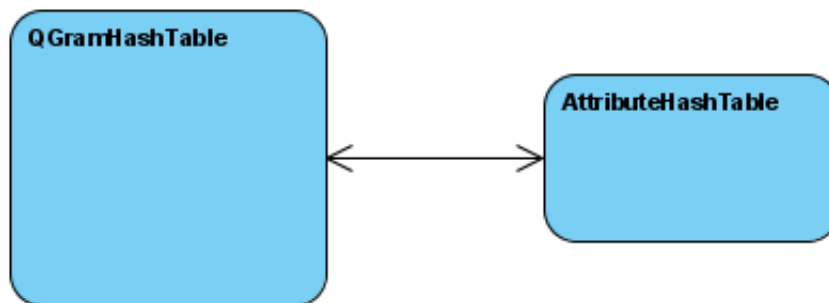


Figure 6.7: Hash table transformation

In the first case there is need to create a new qgram based hash table, obtain all read tuples in sequence, for every tuple obtain the join attributes qgrams and for every qgram create a new entry in the new hash table. In the second case there is need to create the new attribute based hash table, obtain all read tuples in sequence and for every tuple create a new entry in the hash table according to its join attribute. We can therefore see that the cost of the transformation is of the same order of complexity of the cost of a single sshjoin pass. As a consequence during a switch the transformation cost is hidden by a single approximate join pass and it does not compromise the overall performance of the algorithm.

It is important also stress the fact that there is no need to recompute already computed results: The portion of the join result tuples already returned by the old algorithm continue to be part of the result and the portions of the input operators already processed by the old algorithm are not reprocessed again. The execution of the new join algorithm continues so from the very point where the old one was interrupted reading new values from the input operators and calculating new results. Returning to the pipeline architecture in figure 4.1, the switch procedure is invisible to the input and output operators connected to ours: They continue to feed in new tuples and consume new results unaware of the change.

Achieving such an elegant behavior from our component is not as easy as it looks. The execution before the switch should be interrupted in a *safe* point and the new algorithm should begin executing from there. We use as *safe* points the same ones introduced in [EFP06], also called *quiescent states*. The particularity of this points is that the results already produced by the old algorithm can be defined precisely in terms of the inputs read by the operator up to then. The new algorithm could then concentrate only in the rest of the inputs still to be read for producing its results. Method *isQuiescent* introduced in section 3.3 could be used whenever necessary to understand if the algorithm is in a quiescent state or not and therefore make the switch in the appropriate moment.

# Chapter 7

## Experimental results

In this chapter the effectiveness and efficiency of our MAR approach to data quality is tested using synthetic data. Some theoretical results drawn in the previous chapters are now validated in practice. Numerical results are obtained and analyzed by using a newly introduced testbed. This chapter is organized as follows: In section 7.1 the approach for the generation of the synthetic data set, as well as the environment where experiments are run are introduced. Section 7.2 introduces the most important assumptions, according to which our experimental results can be generalized as representing several real-life scenarios. In section 7.3 the effectiveness, and as a consequence reliability, of the probability models introduced in chapter 5 is tested to find the best representative model. Finally, in section 7.4, the MAR approach, with the best probabilistic model according to experiments, is tested to show its effectiveness in several scenarios.

### 7.1 Experimental setting

In order to assess the effectiveness and efficiency of the system in real life scenarios, experiments using real and/or synthetic data should be performed. We chose to use synthetic data for our experiments, but with realistic characteristics, as these were easier to obtain (real data had to be taken from real applications used in real organizations) and easier to control (allowing to better evidence the limitations and advantages of the methodology).

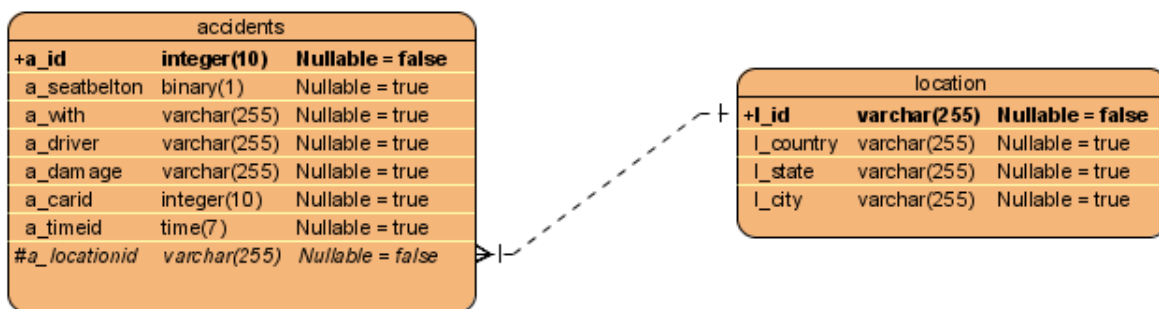


Figure 7.1: The *accidents* and *location* tables

A *data generator* machine has been developed. We contacted Volker Markl at IBM Almaden and asked him to lend us the *DMV* data generator he had implemented and IBM had used to test various products of its suite. The generator was entirely written in *C* language and used a set of text files containing

- world economic important locations,
- car trade marks,

- colors,
- people names,

to generate various data tables representing an accidents database, with accidents occurred, time and location of the accident, drivers involved in the accident, damage caused to the driver etc. For the purpose of our experiments the only important tables are *accidents* and *locations*, reported in Figure 7.1 the former having a foreign key constraint over the latter. We will concentrate only on these tables and use them as the input operators for the join operations performed by our system. The *accidents* table has the following attributes

**a\_id** the unique identifier of the accident, which is also the primary key of the relation,

**a\_seatbelton** a binary value indicating the usage or not of the seat belt by the driver during the accident,

**a\_with** the other object involved in the accident, i.e. another *car*, a *person* or a *tree*,

**a\_driver** the general condition of the driver, i.e. *unharmmed*, *injured* or *dead*,

**a\_damage** the damage caused to the car, i.e. *none*, *minor* or *major*,

**car\_id** the unique identifier of the car involved in the accident,

**a\_timeid** the time of the accident,

**a\_locationid** the location where the accident occurred as a concatenation of the city name, state and country. This attribute serves as a foreign key attribute to the *location* relation.

while the *location* relation has the following attributes

**L\_id** a location name, which is also the primary key of the table,

**L\_country** the location country,

**L\_state** the location state,

**L\_city** the location city.

Originally, only 304 different locations were present in the configuration files. As a consequence only accidents that happened in those locations could be generated. These was a limitation to our adaptive technique that needed more voluminous data, first of all to justify the use of an adaptive strategy on a longer operation and second to give time to the adapt component to monitor and assess the environment before responding.

First, if the inputs involved in the join operation were too small there would not be great difference in execution time between performing an exact join or an approximate one. The difference between different orders of complexity such as quadratic and linear are more evident with large inputs (and less with small ones) [Pap94]. In case inputs were small, the following optimistic approach could also be used and still guarantee brief response times: apply a complete exact join first. If the final result size is acceptable (e.g. equal to the expected one) than execution has finished. If, on the other hand, the result size is lower than the expected one and not acceptable (too many matches have been lost) then perform a complete approximate join over the inputs.

Second, if the inputs involved in the join operation were too small, the use of an adaptive strategy and a whole system to implement it may not be justified. As the join execution would be too short, it would be no time for the system to gather statistically significant monitored parameters, resulting in impossibility to decide if matches are actually being lost or not before the join termination, and

as a consequence to the impossibility to adapt. Even provided that there were enough time to make statistically significant observations and an adaptive decision was made to change the actual join algorithm, the adaptation cost could result too high to justify the effectiveness gain on the rest of the result (big part of the join has already been executed).

In order to obtain more voluminous data the list of cities should be extended with new ones. We chose to discard the original one and use the list of all Italian cities instead. This list contained 8082 entries instead of IBM original 304. As a consequence two sufficiently big tables could be generated as inputs to the join execution

- a *locations* table with 8082 entries,
- an *accidents* table with at list 8082 entries.

Some minor modifications had to be made to the code, to make it compatible to the new cities list.

Once a mechanism to obtain the basic database tables were provided, another one for injecting controlled duplicates on the data had to be introduced. Indeed, if duplicates were never present in the data, there would be no need to worry about missing matches and for data adaptation to be involved. New *Java* code was written to inject duplicates in the data, providing an output table with duplicates given an input clear one, according to the following procedure

1. open the input database table for a reading operation
2. (a) if there are more tuples to be read from the input database, read next tuple  $t$   
(b) else, exit procedure
3. decide if  $t$  is going to be a duplicate or not
  - (a) if  $t$  is going to be a duplicate inject a controlled edit distance error in the join attribute of  $t$
  - (b) else, do nothing
4. write  $t$  in the output database
5. return to step 2

First of all the input database is opened for a reading operation in 1. Then, the tuples in the input database are read in order, until there are no more to be read, at point 2. For every tuple read a decision is made if this is going to be a duplicate or not, according to a given probability parameter  $0 \leq p \leq 1$ . More specifically a random number between 0 and 1 is drawn. If this number is lower than  $p$ , the current tuple is going to be a duplicate, if this number is greater than  $p$  the no error will be injected in the tuple. In case a duplicate decision is drawn, the former join attribute of the tuple, namely  $jA$  is transformed to obtain a new similar one, namely  $jA'$ , such that  $ed(jA, jA') = d \neq 0$ , meaning that the edit distance between the two values is different from 0. The transformation is made with one of the following policies

- a fixed size edit distance,  $d = 1$ ,
- a randomly drawn edit distance by a uniform distribution  $un$  with mean  $m$  and standard deviation  $std$ ,  $d = un(mean, std)$

In both cases insertion, deletion and replacement operations are considered as occurring with the same probability (e.g. all having weight 1).

The error injector procedure was further extended to inject other types of noise (see 4.2) on the data such as

- *heterogeneity* value  $h$
- *data skewness* with parameters  $freq$ ,  $sk_{perc}$
- *order* with value  $order$

The heterogeneity value  $h = z_1 : z_2 : \dots : z_n$  is a list of values between 0 and 1 and such that  $\sum_i z_i = 1$ . Number  $z_i$  in the list represents a contiguous zone of length  $\frac{z_i}{|database|}$  tuples in the database table with homogeneous characteristics such as duplicates percentage, skewness value and order (see section 4.2). The homogeneous case is the one with  $h = 1.0$ .

The data skewness parameter offers the possibility, for each one of the zones defined in  $h$  to contain  $freq$  values for the join attribute that cover  $sk_{perc}$  of the whole zone. So for example if the zone contains 1000 tuples,  $freq = 10$  and  $sk_{perc} = 60\%$ , 60% of the zone will contain 10 *lucky* values for the join attribute, while the rest 40% may contain any of the other possible values.

The order value offers the possibility to lexicographically order the tuples of a given partition  $z_i$  according to their join attribute value as ascending or descending. If this value is not set, than a scrambling (randomizing) procedure will be performed on the data.

In our architecture databases are implemented as binary files and *XXL* [vdBBD<sup>+</sup>01] (eXstensible and fleXbile Library) is used to retrieve data from the files. *XXL* is a high level, easy to use, platform independent, *Java* library supporting the implementation of query functionalities. The independence of the library allows interaction with any type of databases. We chose binary files as there was no need for advanced functionality form well known proprietary or not databases such as *Oracle* or *MySQL*. Binary files instead are easy to use and only an additional trivial operation had to be implemented in order to transform former text files to binary ones used by *XXL*. In Figure 7.2 the data creation process is reported. First of all Volker generator is used to create error-free (and in general noise-free) data. The obtained files are then given in input to the noise injector, having previously set the noise parameters. The text file obtained in output has further to be transformed in binary data, ready to use by *XXL* code embedded in the join algorithms.

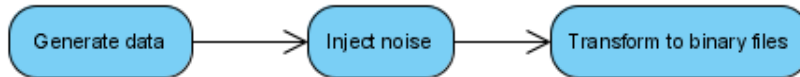


Figure 7.2: The database creation process

To implement the *sshjoin* 3.3 and *shjoin* [WA91] algorithms as well as the whole MAR architecture *Java* programming language, version 6.0, is used.

All experiments are performed on a *Windows XP Service Pack 2* platform, with *Intel Pentium M* 1.73 GHz CPU and a main memory of 1 GB.

## 7.2 Assumptions

In this section various assumptions over the generated data sets are made. These assumptions are further used in sections 7.3 and 7.4 during the experimental validation of the several probability models, introduced in chapter 5, and of the MAR approach based on the specific binomial  $n^{th}$  order distribution model, introduced in 5.2, respectively.

We assume that a foreign key constraint exists between attribute *a\_locationid* of *accidents* table and attribute *Lid*, the primary key of the *location* table (see Figure 7.1). The foreign key constraint is a necessary condition for all probability models described in 5 to be reliable representatives of the join result size variation over time, in duplicates absence. In more general scenarios, where a foreign key condition does not hold, the result size could be estimated by other means such as the

ones introduced in [CR94]. We further assume that there are no null values in attribute *a\_locationid* (*accidents* table). If such null values were present, the problem would be overcome by not considering these values as part of the join result set. For instance, if there were 1000 null values in *accidents*, and no duplicates were present, the join result size observed at the end of a computation would be  $|accidents| - 1000 = 8082 - 1000 = 7082$ . A similar approach could be used by the particular probability model used in the analysis MAR component to estimate result size at execution point  $n$ , as the result at execution point  $n - |nullObserved|$ , where  $|nullObserved|$  is the number of null occurrences observed in the *a\_locationid* attribute to execution point  $n$ . Counting the occurrences of nulls during join execution is straightforward and can be achieved by incrementing a counter every time a null value is seen.

We also assume that the skewness (see section 4.2) present in the data is not relevant, i.e. it is uniform or nearly uniform, with every value of the *a\_locationid* attribute having a similar number of occurrences to all the others in the *accidents* table. This is another necessary condition for the probability models introduced in chapter 5 to hold.

We assume that no order exists on the *accidents* table with related to the *a\_locationid* attribute. This is a realistic assumption in many scenarios in which there is the possibility to make a preliminary randomization process (scrambling) over the whole or part of data before actually performing the join algorithm (see section 4.2).

We assume that the threshold is set in such way, e.g. by the data provider, that no false positives are introduced by the approximate join execution, although we do not make any assumption on total missing of false negatives. Recall that, with related to the approximate join algorithm, given two tuples  $u'$  and  $v'$ , duplicate values of original  $u$  and  $v$

- a false positive is introduced, if  $(u', v')$  is a tuple returned in the join result, but  $(u, v)$  was not a perfect match,
- a false negative is introduced, i.e. a tuple is missing in the join result, if  $(u', v')$  is not part of the result, but  $(u, v)$  is a perfect match.

when with perfect match we mean that the join attributes of both tuples match according to the strict string equality definition.

Intuitively, a false positive occurs if the approximate join, according to a particular similarity function used, introduces a match that should not be part of the result. For example considering *Mark Twain* and *Mark Twein* as a match, assuming that a typographical error occurred during the second value insertion, while the second is in reality a completely different person, would be a false positive. A false negative, on the other hand, intuitively occurs if the approximate join, according to a particular similarity function, fails to recover a match that should be considered as part of the result. For example failing to consider *Mark Twain* and *Mark Twein* as a match, in case the second value is a typographical error of replacing an *a* with an *e*, would be a false negative.

Our assumption of not introducing false positives rely on the fact that the provider knows that quality problems may occur in his data, and that such problems in most of the cases occur with specific threshold characteristics, i.e. a certain edit distance. We then use these thresholds to set parameters of the approximate join algorithm in such a way that no false positives occur, maybe tolerating a certain number of false negatives.

Supposing that duplicates are present only in one of the tables is not a restriction. The idea can be generalized as follows: For the purpose of this thesis the concept of a duplicate as a single value is not important, we focus on the concept of a duplicate in the context of a pair, or a match. To clarify the ideas with an example, it is not important that *john malkovitch* is a duplicate while *john malkovich* is the real representation. The important point is that the pair  $(john\ malkovitch, john\ malkovich)$  will not be a match according to any normal join. As a conclusion, either *john malkovitch* or *john malkovich* could be treated as a duplicate without affecting the algorithm, the important

point is to use an approximate join to classify the couple because at least one of its elements is a duplicate. The *symmetry* property of duplicates inside a pair permits us consider their presence in only one of the tables.

### 7.3 Effectiveness of the probability models

In this section the effectiveness of the different probabilistic models introduced in chapter 5 is tested with duplicates homogeneously distributed in the *accidents* relation, i.e. the distribution of duplicates in the whole relation is uniform, while there are no duplicates present in the *location* relation. As measure of effectiveness we use the time, or more specifically the number of tuples seen by the join algorithm as a proportion of the total number of tuples present in *accidents*, to understand that duplicates presence is resulting in missing matches in the final result set and ring an alarm bell signaling that a switch to exact join is necessary. Intuitively this corresponds to the proportion of join execution in which exact join was used, while an approximate one should be used instead permitting to recover all non-exact matches. The lower is this index, the higher is the effectiveness of the model, as less time is needed to take knowledge of the duplicates presence. For instance, if there are two probability models, the first triggering the switch after 2000 tuples have been read, and the second after 4000, being the cardinality of *accidents* 8082, the effectiveness index of the first is going to be  $2000/8082 \approx 0.25$ , while the effectiveness of the second  $4000/8082 \approx 0.5$ . The first method would in this case be more effective.

The data generator and error injector tool introduced in section 7.1 have been used to obtain three different configurations for the *accidents* table with

1. no duplicates,
2. 5% uniformly distributed duplicates,
3. 10% uniformly distributed duplicates.

While the last two configurations are used to assess the effectiveness of the models, the first one is used to show their behavior in total absence of duplicates. We should expect that in this case none of the models will require to switch from exact to approximate join.

Finally, four probability models have been used to test the previous scenarios as introduced in chapter 5

1. Chebyshev binomial model (or binomial for short),
2. Chebyshev hypergeometrical model (or hypergeometric for short),
3. binomial  $n^{th}$  order distribution model,
4. hypergeometric  $n^{th}$  order distribution model.

In the first two cases the need of a switch is triggered at execution point  $n$ , if the difference between the result size estimated by the model and the observed one is higher than three times the standard deviation

$$\mu_n - rs_n \geq 3\sigma_n \quad (7.1)$$

where  $\mu_n$  and  $\sigma_n$  are respectively the expected result size and the standard deviation at execution point  $n$ , while  $rs_n$  is the observed result size at the same point.

In the third and fourth cases the need of a switch is triggered, if the probability of seeing a certain results size at execution point  $n$  is lower than 0.05, namely the severity threshold

$$P(X_n \leq rs_n) \leq 0.05 \quad (7.2)$$

where  $rs_n$  and  $X_n$  are respectively the observed result size and the random variable representing the result size value at execution point  $n$ .

Figure 7.3 reports the compared effectiveness of the four different models. Notice that in both scenarios the least effective is the Chebyshev binomial one, followed by Chebyshev hypergeometric, then by the binomial  $n^{th}$  order distribution and at the end, by the most effective hypergeometric  $n^{th}$  order distribution model. We see that the experimental results validate the theoretical ones, that we draw in chapter 5 according which

- the hypergeometrical models performed better than the binomial,
- the outlier detection models performed better than the Chebyshev.

Notice also that, the higher presence of duplicates, here as a 10% rate compared to a 5%, the higher the effectiveness of the models. Another observation connected to this is that at a higher duplicates rate the four models tend to separate in two groups, the Chebyshev and the  $n^{th}$  order distribution, with the models inside a group having a very similar effectiveness and the models of different groups having very different effectiveness. In the 5% rate scenario, for instance, all methods have a similar behaviors, while in the 10% one the separation in groups is more visible.

Finally, the computational complexity of the models is inversely proportional to their effectiveness. Chebyshev binomial is the least expensive, followed by Chebyshev hypergeometric. Then, the binomial  $n^{th}$  order distribution model is more expensive than the previous, but less expensive than the hypergeometric  $n^{th}$  order distribution, which is the most expensive of all. The two Chebyshev models are less expensive than the  $n^{th}$  order distribution ones because for the firsts only a difference between e constant and an the expected values (which should be computed) is made, while for the seconds a cumulative probability, which is a large sum of probabilities should be computed (see chapter 5). From experimental results, we see that the binomial  $n^{th}$  order distribution can be a good trade-off between effectiveness and efficiency (computational cost), being visibly more effective than both Chebyshev models, almost the same effective as the hypergeometric  $n^{th}$  order distribution and more efficient than this last model. For the above reasons we use the binomial  $n^{th}$  order distribution model in all experiments of section 7.4.

Figures 7.4–7.15 show more in detail the behavior of the various models with different characteristics for the *accidents* table: no duplicates present, duplicates present homogenically with 5% rate and duplicates present homogenically with 10% rate.

Figures 7.4–7.6 report the cases in which duplicates are not present in *accidents*. Notice that neither Chebyshev binomial model (Figure 7.4) nor  $n^{th}$  order distribution model (Figure 7.6) trigger a switch during the execution. This is also the behavior expected from the models. The hypergeometric model (Figure 7.5) and the hypergeometric  $n^{th}$  order distribution (Figure 7.7) on the contrary do not trigger the switch during almost all the join computation, but erroneously assume that duplicates are present at the very computation end triggers the switch once. This erroneous behavior is due to the high sensibility of the hypergeometric to slight variations in the data characteristics, i.e. skeweness and entropy, as well as to its unreliability for values of the  $n$  parameter (execution point) very near to the cardinality of the *accidents* table.

Figures 7.8–7.11 show the behavior of the models in case 5% duplicates were present in *accidents* from the execution beginning to the execution point in which the switch is triggered. Notice the faster detection of duplicates in the Chebyshev hypergeometric model (Figure 7.9) with respect to the Chebyshev binomial one (Figure 7.8), while both models are inferior to the  $n^{th}$  order distribution models, with the hypergeometric  $n^{th}$  order distribution (Figure 7.11) slightly more effective than the binomial  $n^{th}$  order distribution one (Figure 7.10).

Figures 7.12–7.15 show the behavior of the models in case 10% duplicates were present in *accidents* from the beginning to the point when alarm bell is rang. Notice the slightly faster detection of duplicates presence from the Chebyshev hypergeometric models (Figure 7.13) with respect to Chebyshev

binomial one (Figure 7.12), both being less effective than the  $n^{th}$  order distribution models, triggering the switch in the same execution point (Figure 7.14 and 7.15).

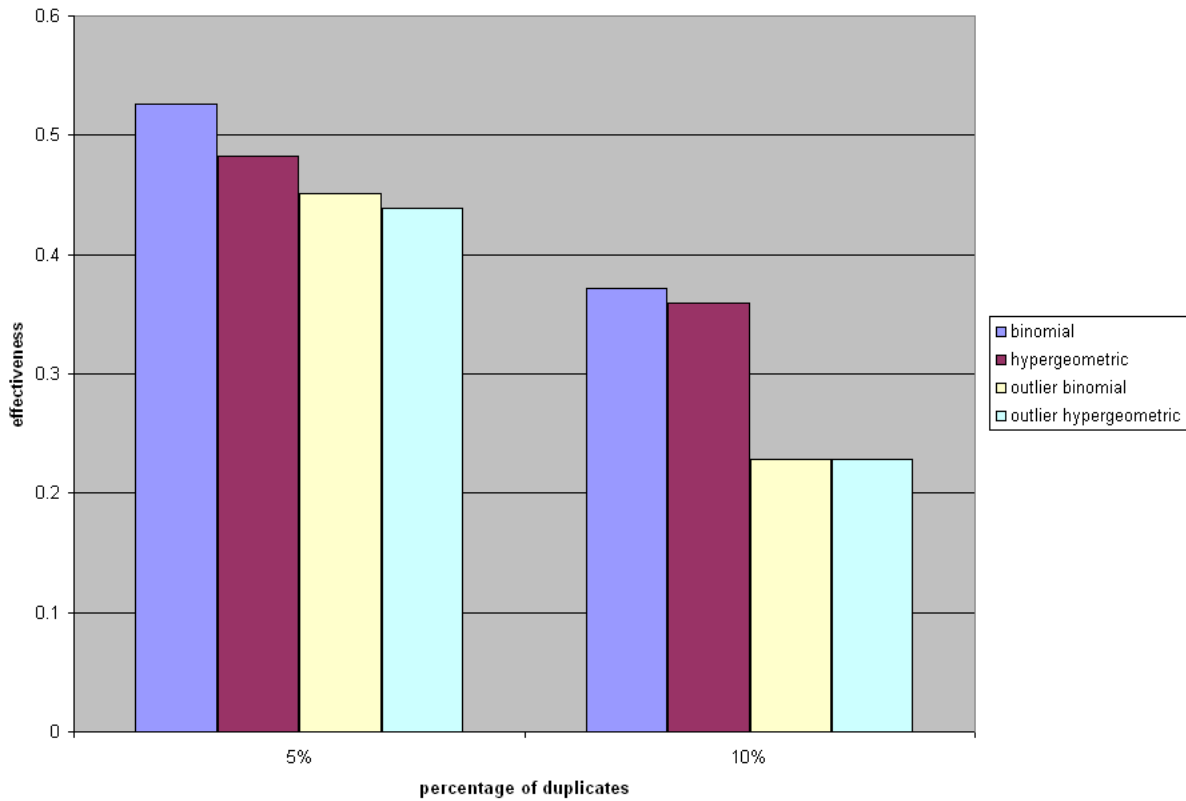


Figure 7.3: Effectiveness of the various models

## 7.4 MAR effectiveness

In this section the effectiveness of the MAR architecture is shown in four specific datasets, which we think are representatives of many real-situation data used by different enterprises. As always we introduce duplicates only in the *accidents* relation, this time with the following patterns (as reported in Figure 7.16)

1. duplicates present homogenically at 5% – 10% rate,
2. duplicates present in 5% and in few large zones,
3. duplicates present in 10% and in few large zones,
4. duplicates present in 10% in many small zones.

In the first scenario duplicates are present uniformly in all *accidents* relation at 10% rate. In the second scenario a few big bursts of duplicates are interleaved by even bigger duplicate-free zones, but the duplicates presence inside each burst is low, i.e. the percentage of duplicates in the whole table is 5. In the third scenario a few big bursts of duplicates are interleaved by even bigger duplicate-free zones, but the duplicates presence inside each burst is high, i.e. 10% in the whole table. Finally, in the third scenario, many small bursts of duplicates are interleaved by many small duplicate-free zones, and even in this case the percentage of duplicates in the whole table is 10.

The MAR architecture, using the binomial outlier detection model, was used to perform the join between *accidents* and *location* tables to measure its effectiveness. An entire exact join as well an

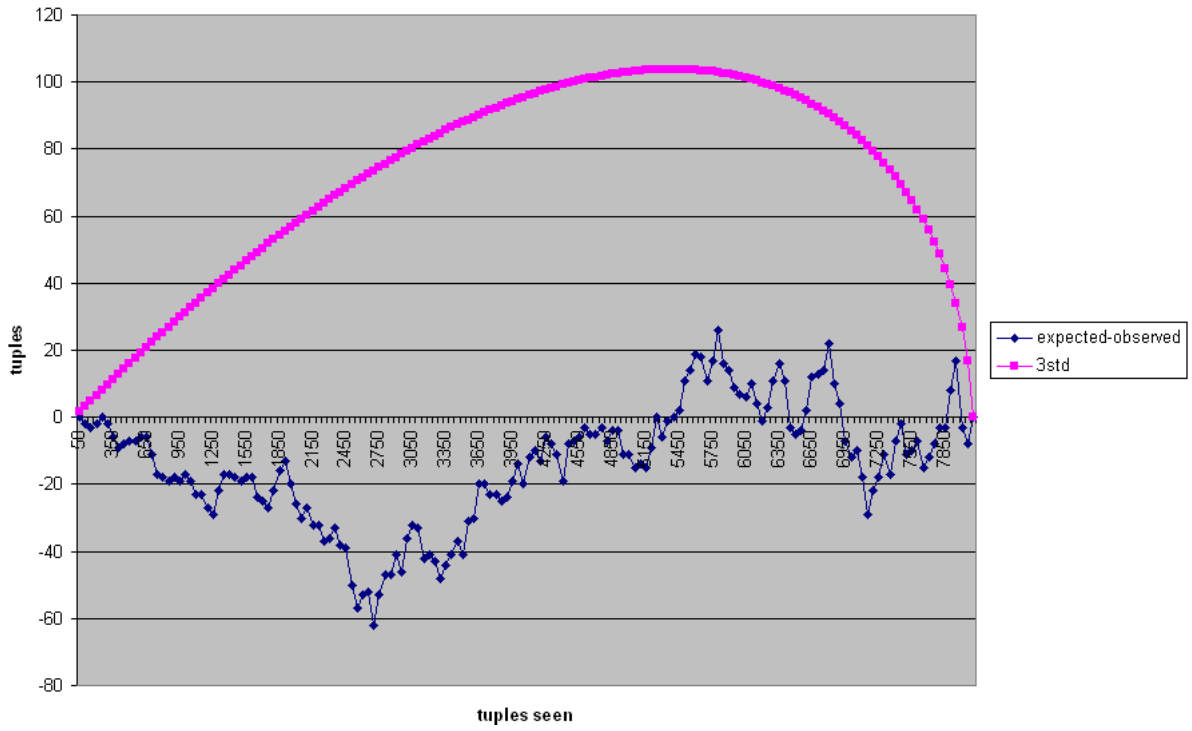


Figure 7.4: The binomial model applied to a no duplicates scenario

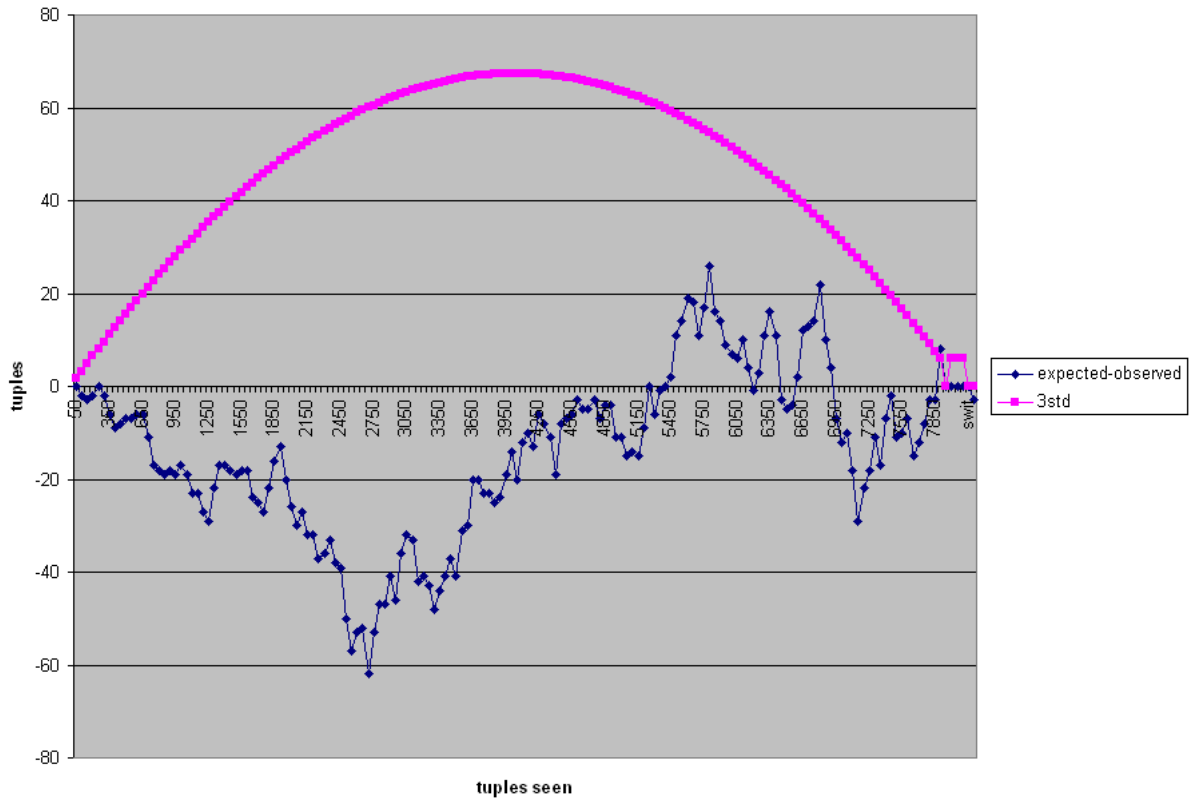


Figure 7.5: The hypergeometric model applied to a no duplicates scenario

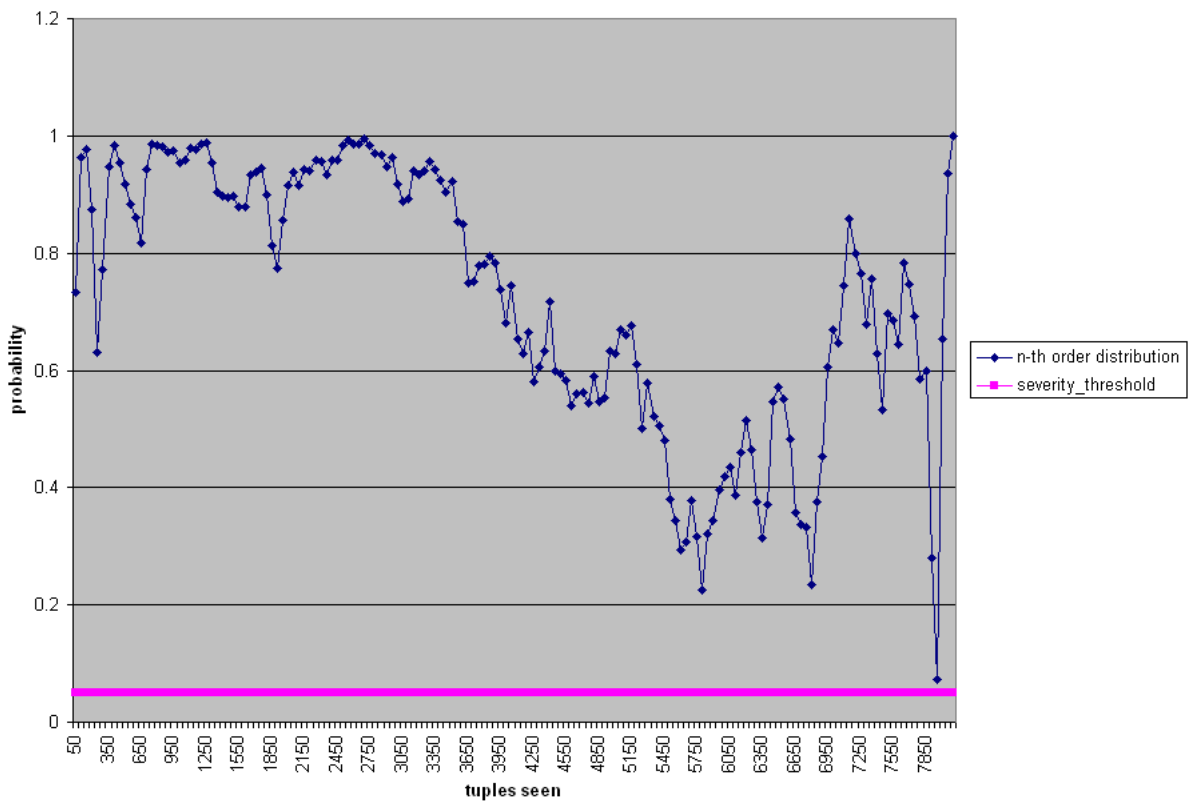


Figure 7.6: The binomial n-th order distribution model applied to a no duplicates scenario

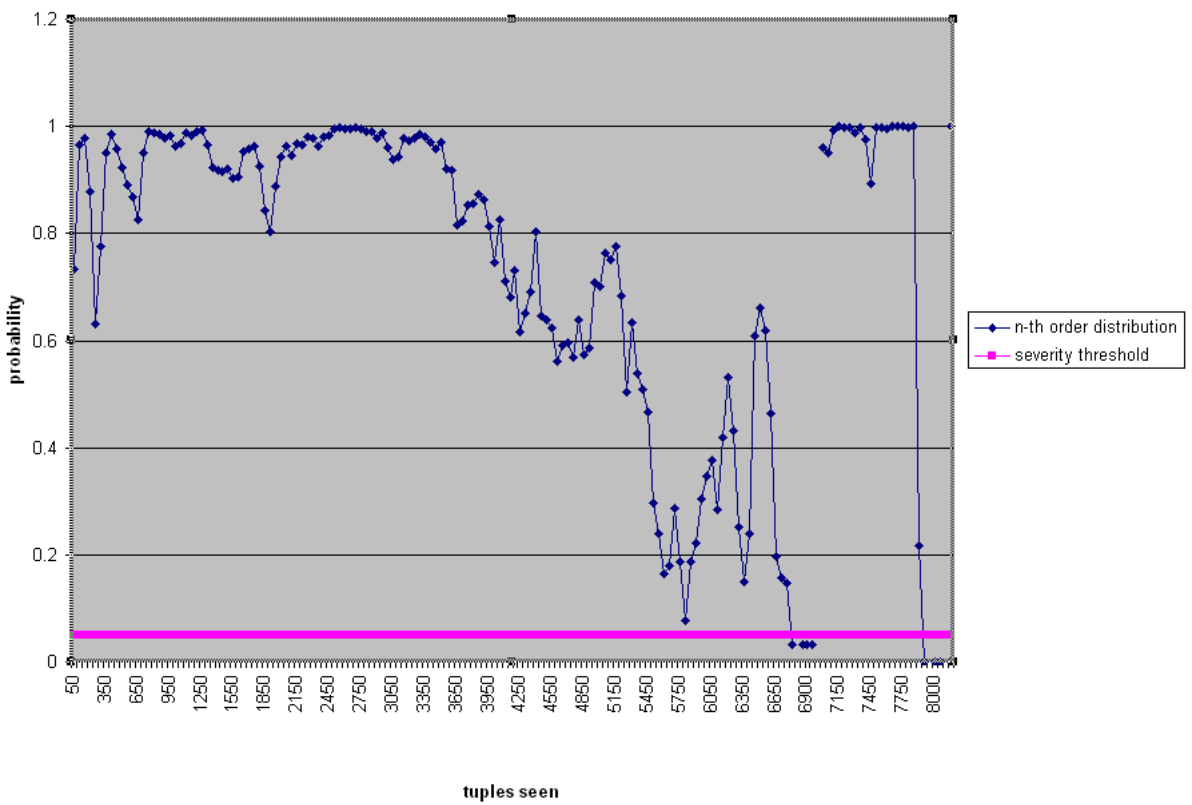


Figure 7.7: The hypergeometric n-th order distribution model applied to a no duplicates scenario

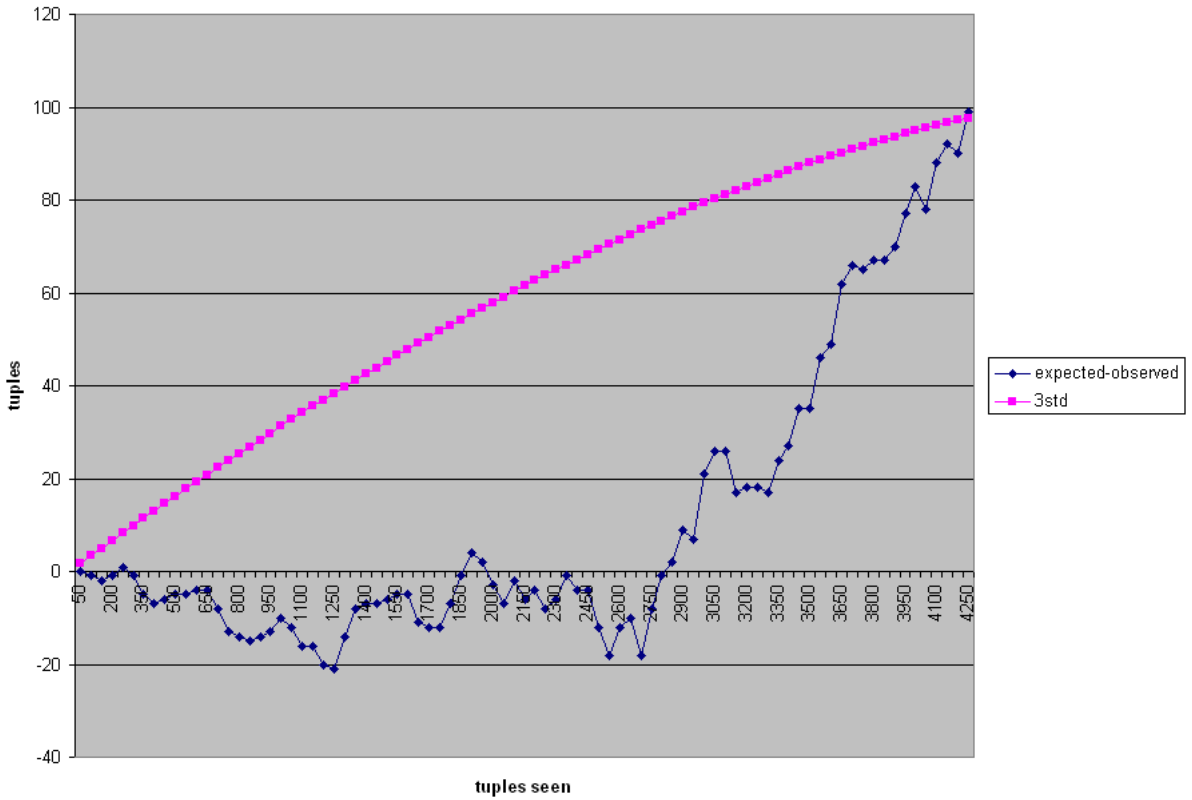


Figure 7.8: The binomial model applied to a 5% duplicates scenario

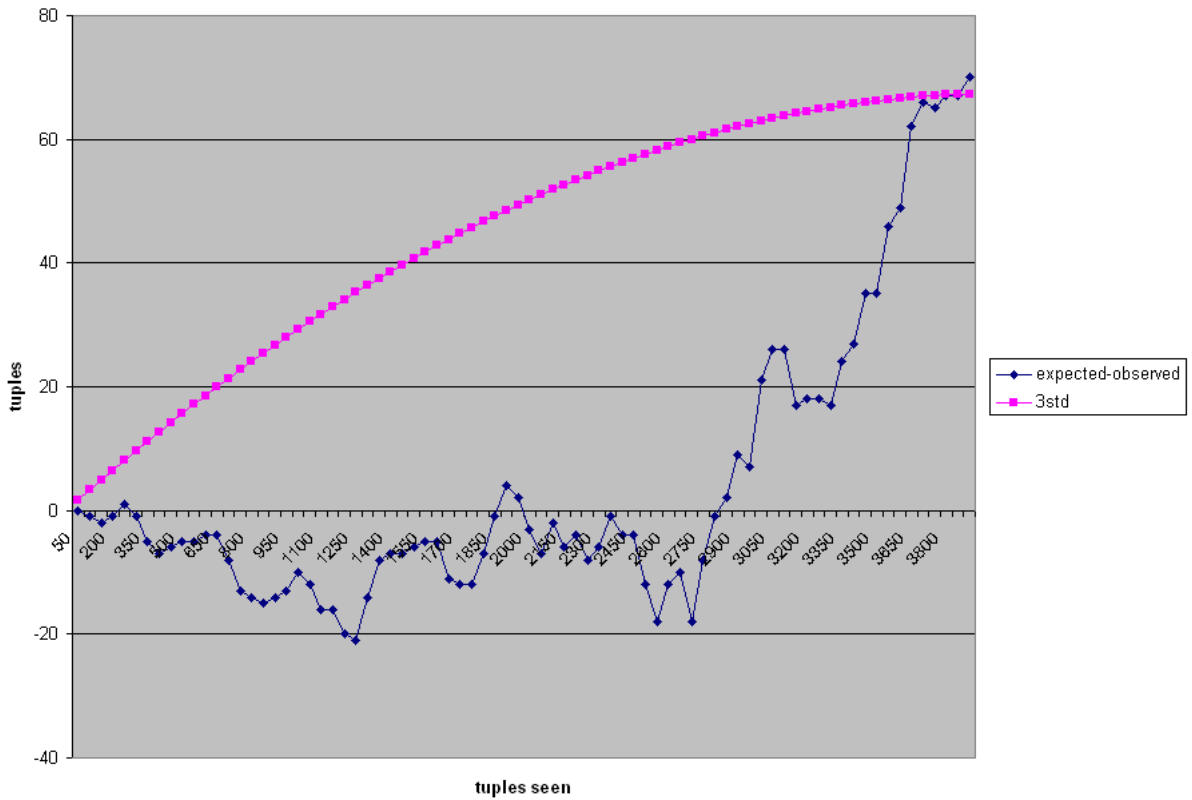


Figure 7.9: The hypergeometric model applied to a 5% duplicates scenario

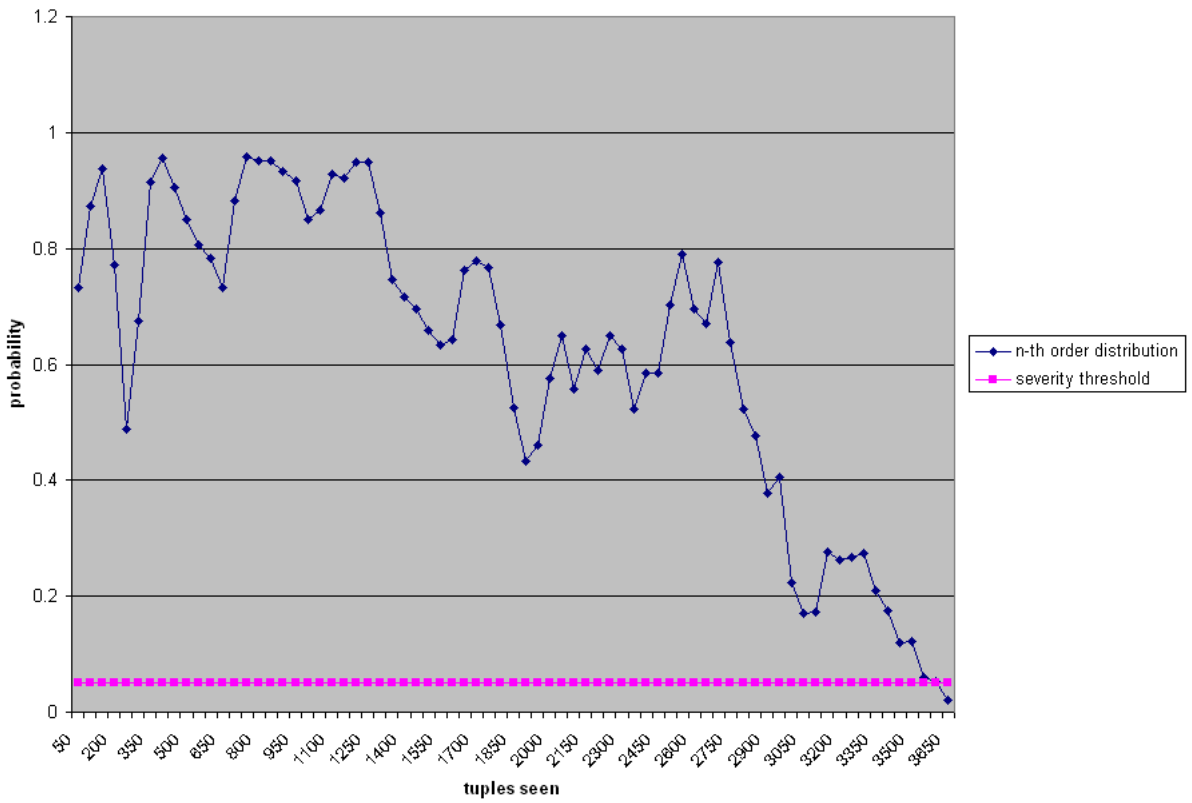


Figure 7.10: The binomial n-th order distribution model applied to a 5% duplicates scenario

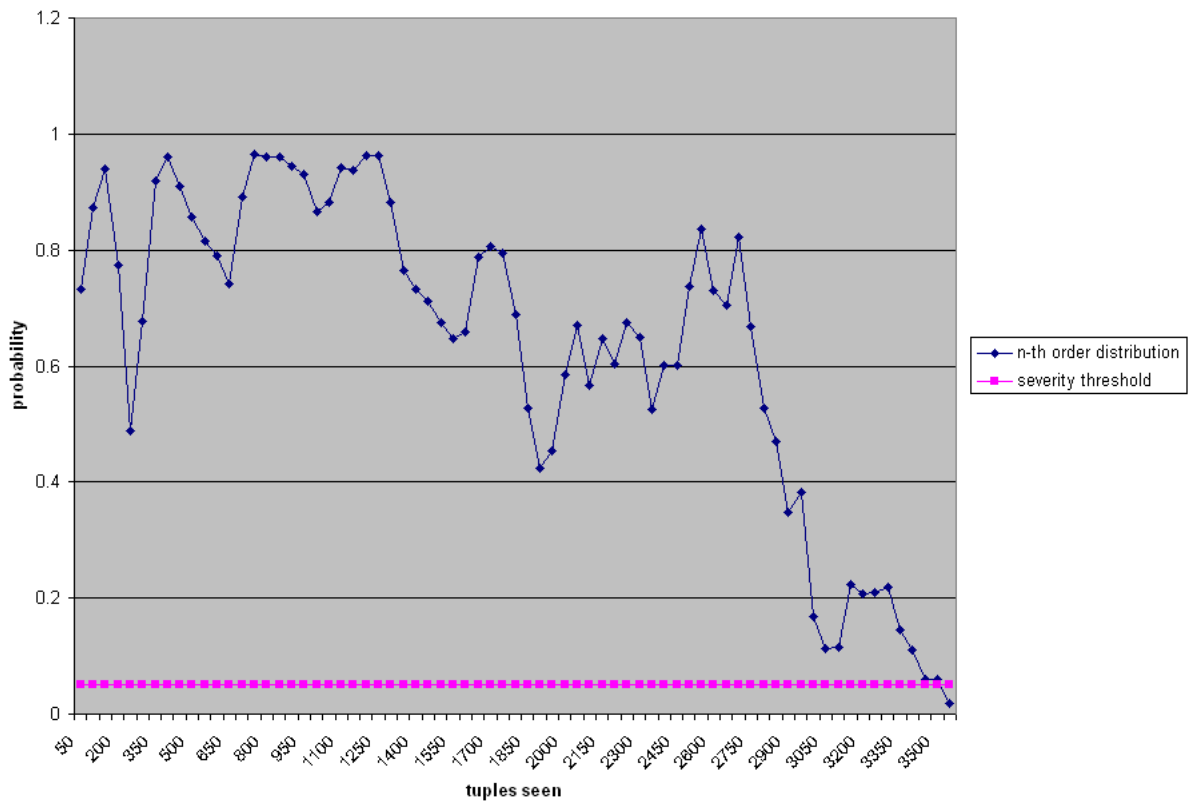


Figure 7.11: The hypergeometric n-th order distribution model applied to a 5% duplicates scenario

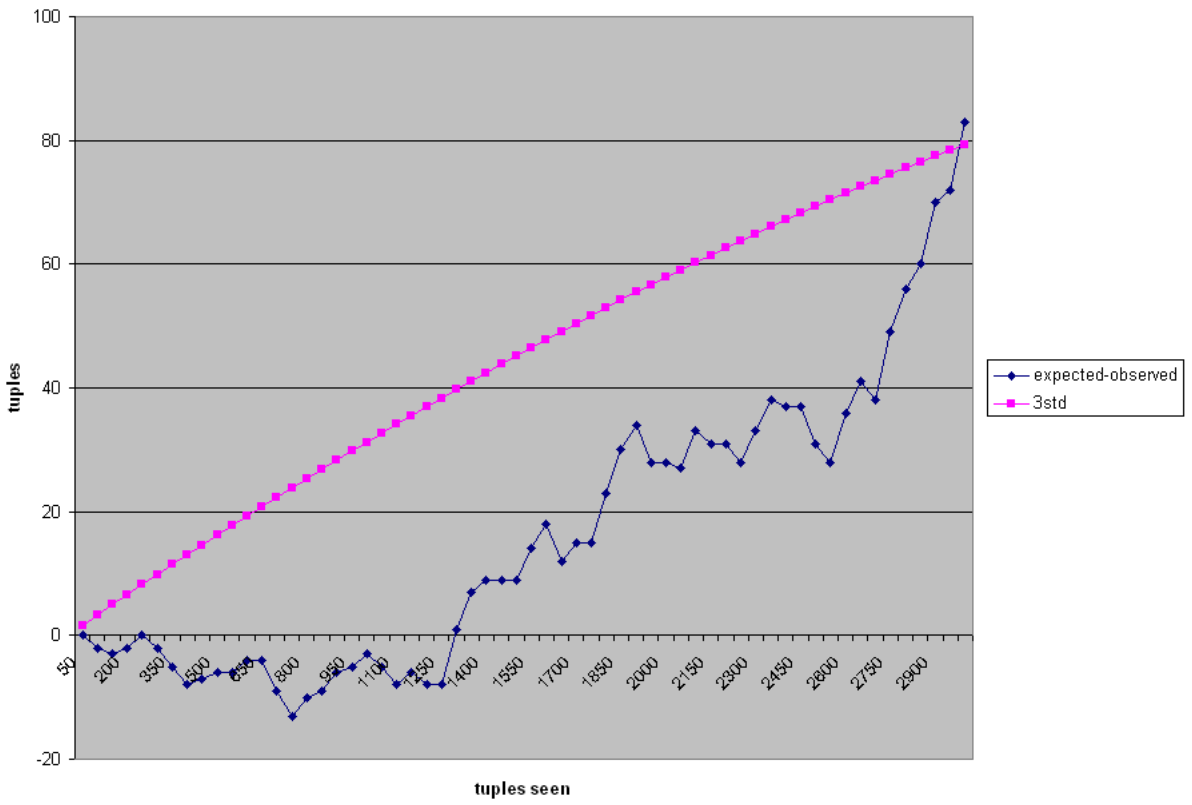


Figure 7.12: The binomial model applied to a 10% duplicates scenario

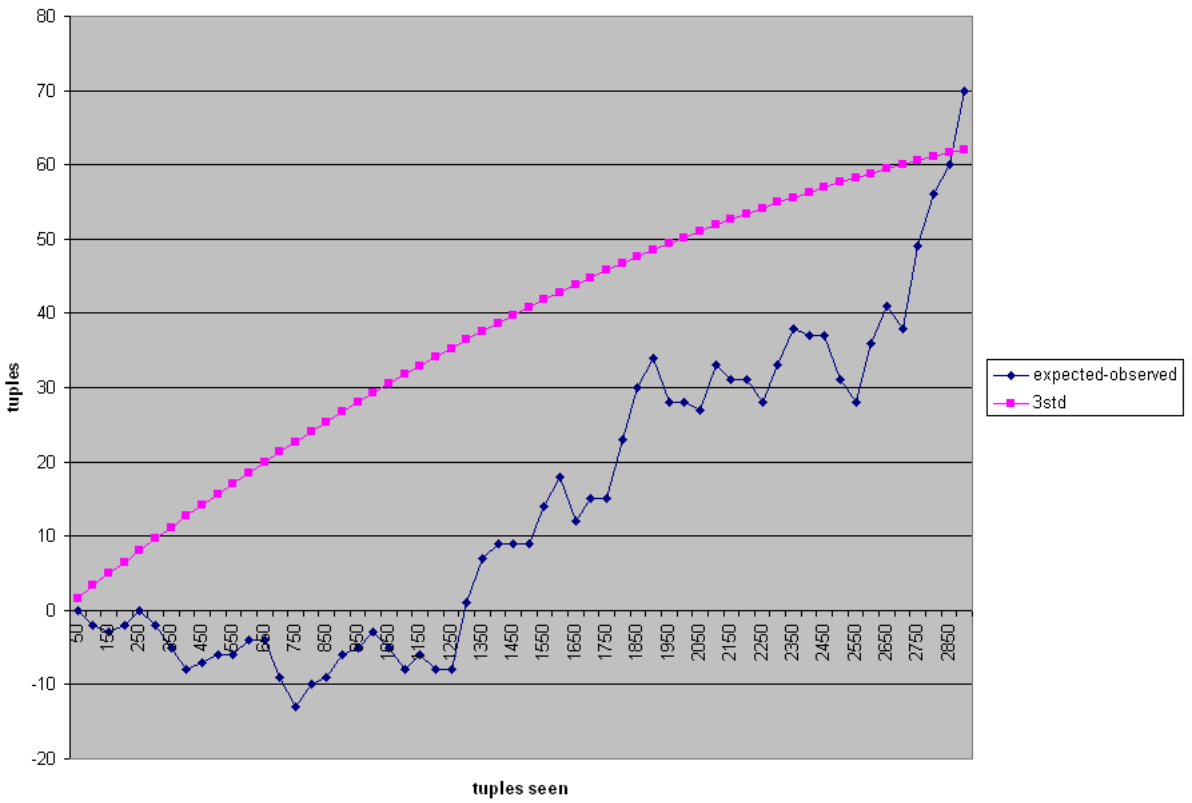


Figure 7.13: The hypergeometric model applied to a 10% duplicates scenario

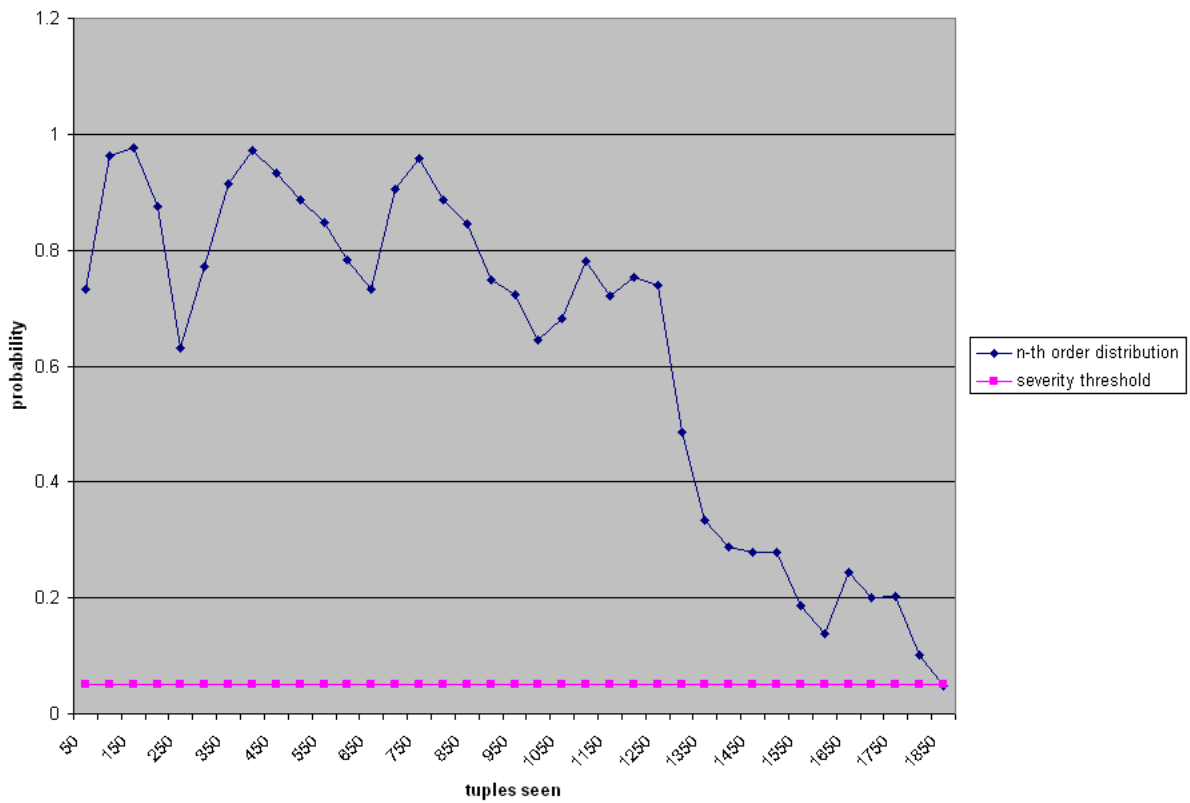


Figure 7.14: The binomial n-th order distribution model applied to a 10% duplicates scenario

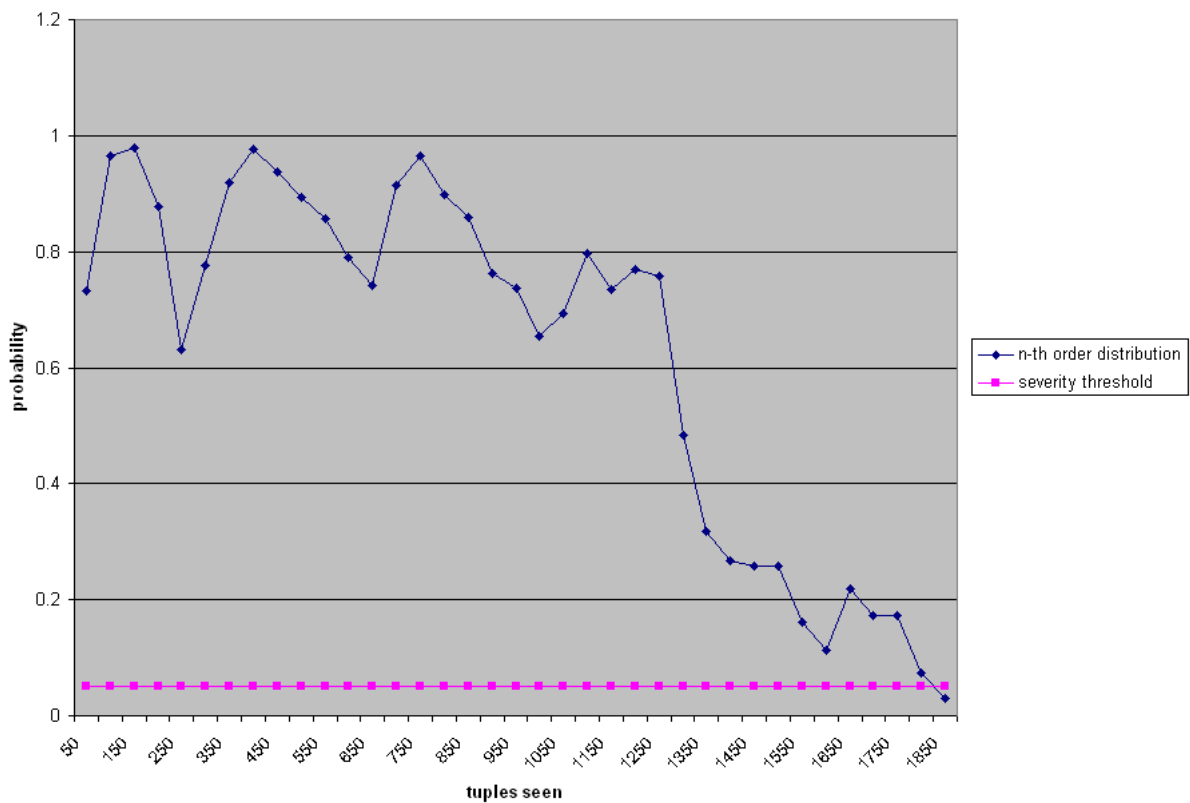


Figure 7.15: The hypergeometric n-th order distribution model applied to a 10% duplicates scenario

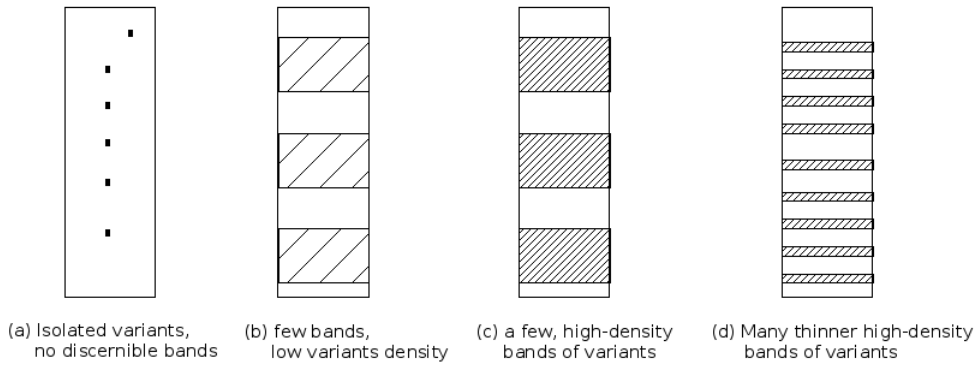


Figure 7.16: Experiments patterns

entire approximate one were then performed on both tables. In both cases the result size of the join was retrieved, namely  $RS_{ex}$  for the exact and  $RS_{ap}$  for the approximate. In case an exact join were performed from the beginning to the end of the computation, all non exact, but semantically correct, matches caused by duplicates presence were lost. In case the approximate join were used from beginning of the computation to its end, the result will be complete and duplicates presence will not affect the result completeness. In case our adaptive MAR approach were used, some matches would result lost from the erroneous usage of the exact join algorithm instead of the approximate one (remember that we use probabilistic approaches to decide and these take time to settle and are not 100% correct) in specific execution intervals. We call  $RS_{MAR}$ , such that  $RS_{ex} \leq RS_{MAR} \leq RS_{ap}$  the result size obtained from the MAR execution. We can now estimate the effectiveness of our adaptive approach as the ratio of approximate matches recovered from its usage compared with both a whole exact and approximate join

$$\frac{RS_{MAR} - RS_{ex}}{RS_{ap} - RS_{ex}} \quad (7.3)$$

This ratio would range between 1 in the best case, corresponding to the usage of an approximate join for the entire execution time, and 0 in the worst case, corresponding to the usage of an exact join for the entire execution time.

Figure 7.17 reports the behavior of the system in the four previously described scenarios, with the worst effectiveness of approximately 65% for scenario 4 and the best for scenario 3, with an effectiveness of approximately 84%. Intuitively the system behaves better in scenarios 2 and 3 because the duplicates burst are big enough and the percentage of duplicates inside a single burst is high enough for them to be noticed from the analysis components in order for appropriate responses to be made (switch to approximate join). In the first scenario, as duplicates percentage is very low in each small portion of the table, it is relatively difficult for the analysis component to incrementally detect their presence and response in time. Finally, in the last scenario, small but violent bursts of duplicates interleaved with small duplicates-free zones make it extremely difficult for the analysis component to understand in time the characteristics of local data, i.e. newly read tuples from the inputs, and hence make the appropriate decision in time.

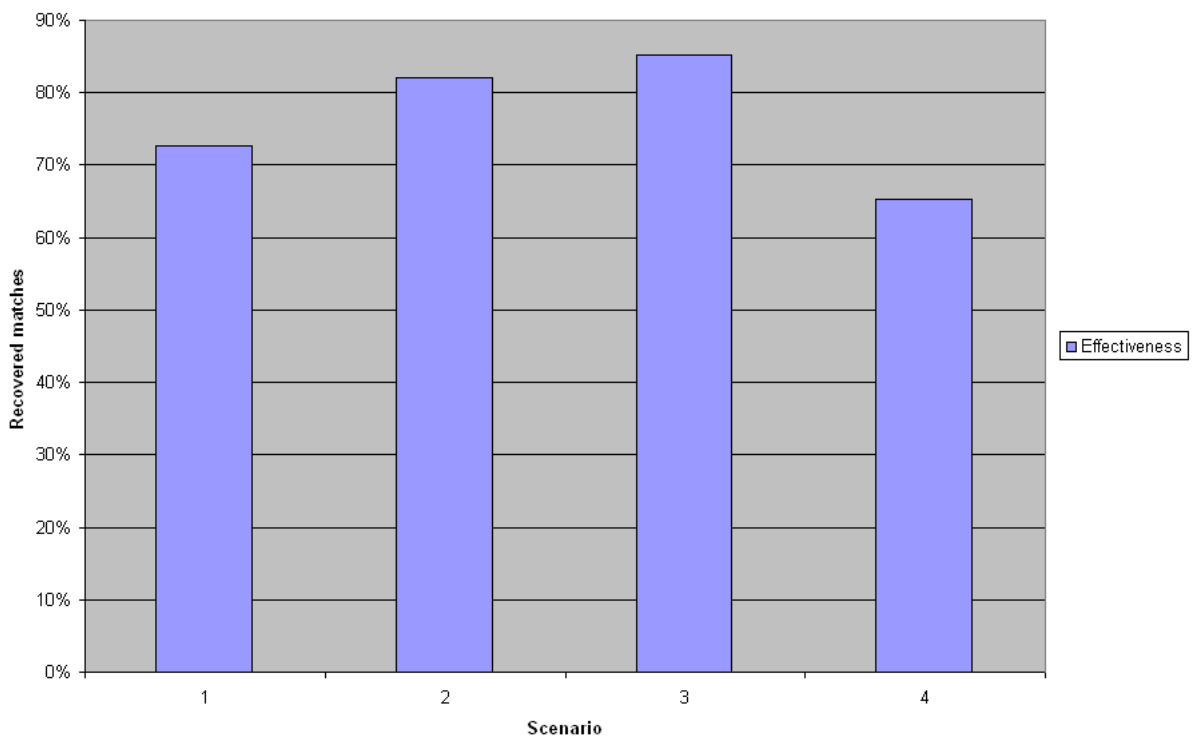


Figure 7.17: The effectiveness of the MAR architecture

# Conclusions

In this thesis we have investigated the well known Quality of Data (QoD) problem of meeting a result completeness constraint during a join execution scenario, even in presence of duplicates. Integration of data delivered from different sources may not always be a straightforward process. Different representations, due to typos or different representation policies, can result in missing matches from the result size that otherwise would be included in it. For instance, consider a scenario in which the same person name is differently represented in two different databases, e.g. *R. Lengu* in the first and *Roald Lengu* in the second. Suppose that the two different databases correspond to two accounts that *Roald Lengu* has opened in two different banks. An integration process, e.g. one that wants to calculate the overall income of *Roald Lengu* for the last year, that does not consider an approximate matching procedure, will consider the two accounts as belonging to different people and would consequently fail to give an exact answer for the income of *Roald Lengu*. On the other hand, an integration process that uses appropriate approximate matching procedures would return the match as part of the result, but would be computationally more expensive.

Previous works, see survey [BS06], used the second integration approach to avoid losing matches from the result. The problem in these cases would be that of introducing a lag between the moment data are obtained and the moment these are made available for integration, not always feasible to afford from real-life applications. In many situations, for instance in data stream integration processes data integration is done on-the-fly and results are used to control real-time systems, e.g. a nuclear plant. In this situations it is inadmissible to perform a unification technique on the data prior to actually integrating it. In other scenarios, such as mash ups to generate dynamic web pages, data may not be stream oriented, but a common user is not going to wait more than 1 minute to obtain satisfactory results. The usage of an expensive approximate join is not justified also in those cases, in which only a small percentage of the inputs contain duplicates. For instance, if the only duplicate entries in the two different databases of the previous example were *R. Lengu*, for the first and *Roald Lengu* for the second, it would be more logical to tolerate such a loss and have minor response times.

As we can see, in many scenarios it results vital to have an acceptable trade-off between effectiveness, i.e. completeness of the result, and efficiency, i.e. response time. In this thesis the desired trade-off is achieved by an adaptive approach that uses the right matching method (join algorithm) in the right moment in order to avoid missing matches

- uses exact join, if duplicates are not actually present in the data,
- uses approximate join, if duplicates are actually present in the data.

where with actually we mean the current portion of tuples being considered by the join as candidates for new matches.

We introduce a new incremental approximate algorithm, namely *sshjoin* (similarity set hash join) to use as our approximate join method and use the standard symmetric hash join [WA91] as the exact one. The *sshjoin* incremental property states that it can be easily used in a pipeline of operators, where input tuples are read one at a time from the output of other pipelined operators, and results are returned one at a time as input to other pipelined operators as well. This property allows the adaptive component to stop join execution at certain points in time, called *quiescent* states, replace

the specific join operator with another one, e.g. an exact join with an approximate join, continue execution from the interruption point without having to recompute results and still guarantee the soundness of the final result set.

We instantiate a MAR architecture for our adaptive strategy, that at every execution point in time decides which join method to use in order to avoid missing matches. The MAR acronym stands for

**Monitor** the join algorithm actual performance and behavior in order to obtain valuable statistics and other type of information,

**Analyze** the previously collected statistics and information to detect problems and/or opportunities,

**Respond** with a new join algorithm in order to solve current problems or exploit in the best way the new opportunities.

We then introduce several probabilistic models to be used by the analyze component in order to estimate the presence or not of duplicates in the join inputs. The model uses the information gathered from the monitor component to estimate such probability. According to the drawn probability a response is made if considered necessary. For instance, if the probability of actually having duplicates is higher than 95% a decision to use the approximate join could be drawn to avoid missing matches. On the other hand, if the probability of losing duplicates were smaller than 95%, e.g. 50%, a decision to use an exact join could be drawn instead.

After introducing our solution, we further implement it and use synthetic but with characteristics similar to real data in order to estimate the effectiveness of the system, i.e. the number of approximate matches recovered with respect to the exact join. An experimental comparison between the several probability models is made as well, to find the one with better trade-off between effectiveness, e.g. how correctly the probability of missing matches is estimated, and computational efficiency.

To our knowledge, this thesis is the first work that applies adaptive query processing (AQP) technique to solve data quality problems. We concentrate on the single record accuracy and the join result size completeness problems. The problems are then completely solved in a particular scenarios such as that of a join between two relational tables, the first having a foreign key constraint in the second and where the data have same particular characteristics, i.e. no skewness is present (all values occur with same frequency) and the entropy is relatively high (data is randomized).

The main contributions of this thesis are: `sshjoin`, a new incremental approximate algorithm for performing join operations in the presence of duplicates; a novel, generic AQP approach to QoD management; an instantiation of that generic approach in which an adaptive join strategy is used to manage result completeness; the tailoring of a set of probability models for the detection of duplicates in join input streams together with a theoretical and experimental confront of their effectiveness; a cost-benefit analysis of the contributed adaptive join strategy by means of a set of experimental results that characterize the circumstances under which, for various types of duplicate-ridden inputs, the adaptive join strategy proves effective and efficient.

In order to show the potential of our adaptive approach, a series of fixed-value parameters used for the decision making process were appropriately tuned allowing the component to reach the best trade-off between effectiveness and efficiency. In the future, the adaptive approach could be extended to such parameters as well, absolving a probable system administrator from the task of manually tuning their values.

In this thesis two specific join algorithms, the exact symmetric hash join and the approximate `sshjoin` one, are used as the only two interchangeable components of the adaptive strategy. In the future, the set of algorithms could be extended with further ones, e.g. merge or indexed loop joins, in order for the adaptive component to completely control the efficiency, and not only the effectiveness, of the approach by using techniques like the ones described in [EFP06].

In the future, the approach could be extended to scenarios in which constraints such as the foreign key, the absence of skewness and the high entropy are not necessary. The problem of integrating two different data sets coming from two different data sources will then be completely resolved by an adaptive approach, by using an approximate join only if necessary, and an exact one otherwise. The approach could then be integrated with little effort in existing databases, datawarehouses and web service systems to allow service providers deliver high quality data to their users maintaining the response time as low as possible.

Finally, we believe that the result completeness of a data integration process between two different, and most probably heterogeneous data sources, is not the only QoD problem that can benefit from an adaptive approach. Other QoD problems, e.g. various currency and consistency ones, could be partially or completely solved in the future by using adaptive strategies.

# Bibliography

- [ACKM03] G. Alonso, F. Casati, H. Kuno, and V. Machiraj. *Web Services: Concepts, Architectures and Applications*. Springer-Verlag, 2003.
- [AGK06] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. *VLDB*, pages 918 – 929, 2006.
- [BGMZ97] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *World Wide Web Conference*, page 391404, 1997.
- [BMC<sup>+</sup>03] M. Bilenko, R. J. Mooney, W. W. Cohen, P. Ravikumar, and S. E. Fienberg. Adaptive name matching in information integration. *IEEE Intelligent Systems*, 18:16–23., 2003.
- [BS06] C. Batini and M. Scannapieco. *Data quality: Concepts, methodologies and techniques*. Springer, 2006.
- [BWPT98] D. P. Ballow, P. Wang, H. Pazer, and G.K Tayi. Modeling information manufacturing systems to determine information product quality. *Management Science*, 44:4, 1998.
- [C<sup>+</sup>03] S. Chandrasekaran et al. Telegraphcq: Continuous dataflow processing for an uncertain world. *CIDR*, 2003.
- [CGK06] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. *ICDE*, 2006.
- [CGM05] S. Chaudhuri, V. Ganti, and R. Motwani. Robust identification of fuzzy duplicates. *ICDE*, pages 865–876, 2005.
- [CR94] C. M. Chen and N. Roussopoulos. Adaptive selectivity estimation using query feedback. *SIGMOD*, pages 161–172, 1994.
- [DF06] M. Denny and M. J. Franklin. Adaptive execution of variable-accuracy functions. *VLDB*, pages 547–558, 2006.
- [DIR07] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1:1–140, 2007.
- [DLR77] A. Dempster, A. N. Laird, and D. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society series B* 29, 1, 1977.
- [EFP06] K. Eurviriyankul, A. A. A. Fernandes, and N. W. Paton. A foundation for the replacement of pipelined physical join operators in adaptive query processing. *EDBT workshops*, 2006.
- [EIV03] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Robust and efficient fuzzy match for online data cleaning. *SIGMOD*, pages 313–324, 2003.

- [EIV07] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING*, 19:1–16, 2007.
- [FS69] I. P. Fellegi and A. B. Sunter. A theory for record linkage. *Journal of the american statistical association*, 64, 1969.
- [G<sup>+</sup>01] H. Galhardas et al. Declarative data cleaning: Language, model, and algorithms. *VLDB*, page 371380, 2001.
- [Goe04] M. Goerk. Sap ag data quality@sap: An enterprise wide approach to data quality goals. 2004. In CAiSE workshop on data and information quality.
- [Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)*, 25:73–169, 1993.
- [Hsu97] H. Hsu. *Probability, random variables and random processes*. McGraw-Hill, 1997.
- [Ins] Data Wharehousing Institute. Data quality and the bottom line: Achieving business success through a commitment to high quality data. <http://www.dw-insitute.com/>.
- [JAF<sup>+</sup>06] S. R. Jeffery, G. Alonso, M. J. Franklin, W. Hong, and J. Widom. Declarative support for sensor data cleaning. *VLDB*, pages 83–100, 2006.
- [JGF06] S. R. Jeffery, M. Garofalakis, and M. J. Franklin. Adaptive cleaning for rfid data streams. *VLDB*, pages 163–174, 2006.
- [KC03] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *computer*, 36:41–50, 2003.
- [LWY07] C. Li, B. Wang, and X. Yang. Vgram: Improving performance of approximate queries on string collections using variable-length grams. *VLDB*, pages 303–314, 2007.
- [MF04] Z. Michalewicz and D. B. Fogel. *How to solve it, modern heuristics*. Springer, second edition, 2004.
- [omb] Office of management and budget. Information quality guidelines for ensuring and maximizing the quality, objectivity, utility, and integrity of information disseminated by agencies. <http://www.whitehouse.gov/omb/fedreg/reproducible.html>.
- [ora] Oracle. <http://www.oracle.com/solutions/bussiness-intelligence>.
- [Pap94] C. H. Papadimitriou. *Computational complexity*. Addison Wesley, second edition, 1994.
- [Par03] European Parliament. Directive 2003/98/ec of the european parliament and of the council of 17 november 2003 on the re-use of public sector information. *Official journal of the european union*, 2003.
- [S<sup>+</sup>92] C. E. Sarndal et al. *Model Assisted Survey Sampling*. Springer-Verlag New York, Inc. (Springer Series in Statistics), 1992.
- [SK04] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. *SIGMOD*, pages 743–754, 2004.
- [Sta95] R. Stanton. Rational prepayment and the valuation of mortgage-backed securities. *In Review of Financial Studies*, 8, 1995.

- [vdBBD<sup>+</sup>01] J. van den Berchen, B. Blohsfeld, J. P. Dittrich, J. Kramer, T. Schafer, M. Schneider, and B. Seeger. Xxl: a library approach to supporting efficient implementations of advanced database queries. *VLDB*, 2001.
- [WA91] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. *PDIS*, pages 68–77, 1991.
- [web] Ibm. <http://www-306.ibm.com/software/websphere/>.
- [WMB99] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishing, 1999.