
INDICE

Introduzione	1
Capitolo 1 Lo stato dell'arte.....	11
1.1 WEKA	12
1.2 Clementine.....	17
1.3 SQL Server	21
1.4 AJAX.....	23
1.5 Considerazioni finali.....	28
Capitolo 2 DPL: un formalismo per la preparazione dei dati	31
2.1 La qualità dei dati	32
2.1.1 La pulizia dei dati.....	34
2.1.2 Problemi di dati “sporchi”	36
2.2 Requisiti del sistema.....	41
2.3 Introduzione agli operatori DPL.....	45
2.3.1 Il costrutto di riscrittura.....	45
2.3.2 La marcatura.....	46
2.3.3 La divisione.....	47
2.3.4 Lo split	47
2.3.5 La fusione.....	48
2.3.6 Rinominazione, rimozione, creazione di un attributo	48
2.3.7 Il ripiegamento	49
2.3.8 Ordinamento per frequenza.....	49
2.3.9 La marcatura e la fusione di duplicati.....	50
2.3.10 Il campionamento.....	50
2.3.11 La normalizzazione	51
2.3.13 La discretizzazione.....	51
2.3.14 I costrutti SQL.....	52
2.4 Introduzione alle espressioni regolari.....	52
2.5 Esempi	56
2.5.1 Applicazione delle regole di riscrittura.....	56
2.5.2 Esempi di trasformazioni di schema	59
2.5.3 Rilevazione dei duplicati.....	62
2.5.4 Integrazione di sorgenti eterogenee	64
2.6 Sintassi.....	68
Capitolo 3 Semantica	77
3.1 La semantica del linguaggio	77
3.2 Le tabelle estese.....	78
3.2.1 Il tipo ‘TabEstesa’	78
3.2.2 Il dominio delle <i>tabelle estese</i>	80
3.3 Nozioni di base per l'implementazione	83
3.3.1 Operazioni derivate frequenti.....	84
3.3.2 Le trasformazioni “locali”	90
3.3.3 Semantica delle espressioni regolari	92
3.4 Implementazione.....	99

3.4.1 Le regole di riscrittura	99
3.4.2 La regola di riscrittura <i>ruleS</i>	100
3.4.3 La regola di riscrittura <i>ruleM</i>	106
3.4.4 La regola di riscrittura <i>ruleT</i>	108
3.4.5 Le trasformazioni di schema	112
3.4.6 DIVIDE	114
3.4.7 SPLIT	118
3.4.8 MERGE	122
3.4.9 RENAME	124
3.4.10 REMOVE	125
3.4.11 NEWATTRIBUTE.....	126
3.4.12 FOLD.....	129
3.4.13 MARK	132
3.4.14 La discretizzazione	136
3.4.15 La normalizzazione.....	159
3.4.16 Il campionamento	168
3.4.17 Ordinamento per frequenza	177
3.4.18 I duplicati.....	179
3.4.19 Operatori SQL	184
3.4.20 Restrizione WHERE.....	188
Capitolo 4 Proprietà del linguaggio	191
4.1 Alcune proprietà fondamentali.....	191
4.2 Le trasformazioni locali	196
4.2.1 Prop. 1: Equivalenza tra sequenzializzazione ed annidamento	197
4.2.2 Prop. 2: Distributività delle trasformazioni locali rispetto all'unione	199
4.3 Le trasformazioni locali di schema	202
4.3.1 Prop. 3: Equivalenza tra composizione ed annidamento.....	202
4.3.2 Prop. 4: Distributività rispetto all'unione	203
4.4 Proprietà della normalizzazione	204
4.4.1 Prop. 5: Commutatività	204
4.4.2 Prop. 6: Equivalenza tra sequenzializzazione ed annidamento	211
Capitolo 5 Esempio: Preparazione dei log all'analisi di "clickstream".....	213
Capitolo 6 Conclusioni	225
Ringraziamènc	228
Appendice A: Le operazioni elementari sulle sequenze.....	229
Bibliografia.....	239

Indice delle figure

Figura a:	Il processo di KDD secondo Fayyad, Piatetsky-Shapiro e Smyth.....	9
Figura b:	Una seconda visione del processo di KDD.....	9
Figura c:	Il processo di KDD e dei suoi stadi secondo J. Han e M. Kamber.....	10
Figura 1.1:	La tabella ed il file in formato ARFF corrispondenti alla relazione PlayTennis	14
Figura 1.2 :	Interfaccia grafica di WEKA	16
Figura 1.3:	Ambiente grafico di Clementine: il desktop e la paletta degli oggetti.....	19
Figura 1.4:	I tipi di trasformazione di AJAX ed un processo di pulizia di dati riferito all'esempio Telecom.....	25
Figura 2.1:	Integrazione di diverse sorgenti.....	64
Figura 3.1:	Una tabella estesa.....	78
Figura 3.2:	Schema logico della implementazione di una trasformazione "locale".....	91
Figura 3.3:	I metodi di campionamento	171
Figura 5.1:	Integrazione di pagine mancanti	217

Introduzione

Negli ultimi venti anni la disponibilità di dati ed informazioni digitali è cresciuta vertiginosamente grazie alla progressiva ed inesorabile diffusione dei computer e al drastico abbassamento del rapporto tra prezzo e capacità dei supporti di memorizzazione. Infatti non è insolito che un sistema di supporto alle decisioni (DSS) contenga milioni o anche centinaia di milioni di record di dati. È dunque facile intuire come la velocità di crescita dei dati vada aldilà della capacità di controllo umana.

I sistemi classici di memorizzazione dei dati, i SGBD (sistemi di gestione per basi di dati), offrono un'ottima possibilità di memorizzare ed accedere ai dati con sicurezza, efficienza e velocità, ma non permettono un'analisi per l'estrazione di informazioni, utili come supporto alle decisioni. Negli ultimi anni sono apparse tecnologie in grado di svolgere questo compito, grazie al contributo di diverse discipline tra cui: l'Intelligenza Artificiale, le Reti Neurali, la Statistica, l'Apprendimento Automatico. Queste tecnologie, applicate alle basi di dati, sono conosciute con il termine **Knowledge Discovery in Databases** (KDD), inteso come un processo per l'estrazione di informazioni, o meglio conoscenza, da grosse quantità di dati. Una delle definizioni¹ di KDD più note è quella usata in "Advances in Knowledge Discovery and Data Mining", 1996, da Fayyad, Piatetsky-Shapiro e Smyth [FPS96]:

“processo non banale di identificazione di pattern validi, nuovi, potenzialmente utilizzabili e comprensibili fra i dati”.

Se analizziamo brevemente tale definizione, possiamo intuire come il termine *processo* indichi che l'estrazione di conoscenza è iterativa, oltre ad essere *non banale*, perché va oltre il calcolo di aggregati sui dati, ed è in grado di indurre informazioni da quelle racchiuse implicitamente nei dati. I modelli (*pattern*) cercati devono

¹ La definizione di KDD più simpatica tra quelle incontrate: “*You torture the data until they confess ...*”

rappresentare qualcosa di nuovo, che non è ancora noto, devono essere generalizzabili anche per il futuro (*validi*), servono a formalizzare la conoscenza e, allo stesso tempo, a stimolare l'intuizione umana (*comprensibili*), e soprattutto, la loro scoperta dovrebbe portare un certo vantaggio (*potenzialmente utilizzabili*).

Il processo di KDD, ovvero il *processo di estrazione di conoscenza*, è costituito da un numero di stadi interattivi, che manipolano e trasformano i dati per riuscire ad estrarre delle informazioni utili. Usama Fayyad, Piatetsky-Shapiro e Smyth [FPS96] identificarono cinque passi nel processo di estrazione di conoscenza (vedi fig. a), e tra questi, lo stadio del **data mining** costituisce l'attività centrale nel processo. In questa fase vengono applicati gli algoritmi per l'esplorazione e lo studio dei dati. Con una certa frequenza, in letteratura, i termini *KDD* e *data mining* sono usati come sinonimi; in questa presentazione si intenderà per *data mining* solo un passo specifico all'interno del processo, identificato dall'insieme delle tecniche, degli strumenti e degli algoritmi usati per esplorare grandi quantità di dati, al fine di individuare informazioni significative, e per la presentazione dei risultati.

In questi ultimi anni, presso il Dipartimento di Informatica dell'Università di Pisa è stato sviluppato un ambiente generale di sviluppo per applicazioni di estrazione di conoscenza dai dati. Tale sistema, denominato **KDDML-MQL (KDD Markup Language – Mining Query Language)**, è nato con l'obiettivo principale di integrare le varie fasi del processo KDD. Si è riconosciuta innanzitutto la necessità di rappresentare, in modo uniforme, i dati in ingresso, i meta-dati, i modelli di conoscenza, e gli stessi problemi di data mining. Questo requisito, indispensabile per fornire l'interoperabilità tra le diverse fasi del processo, così come tra i diversi modelli di conoscenza, è stato soddisfatto attraverso l'utilizzo di un linguaggio standard per la rappresentazione e lo scambio dei dati: il linguaggio di mark-up **XML**. Esso costituisce la base per la realizzazione del linguaggio di mark-up **KDDML** del sistema, il quale è interamente implementato in Java. In un secondo momento è stato definito e integrato nel sistema un nuovo linguaggio di interrogazione, che rendesse l'utilizzo dell'ambiente KDDML più semplice. Tale linguaggio, denominato **Mining Query language (MQL)**² si pone ad un livello di astrazione superiore, conservando la flessibilità, l'estendibilità e l'espressività tipiche del linguaggio KDDML. Tramite esso è possibile specificare un problema di estrazione di conoscenza come un processo di interrogazione sui dati. Infatti, il

² Il linguaggio MQL è stato sviluppato nella tesi [Tesi2]

linguaggio permette di combinare dati e modelli di conoscenza (regole di associazione, alberi di classificazione, clusters) attraverso degli operatori secondo uno stile algebrico. Il risultato di un'operazione può costituire l'ingresso di un altro operatore, realizzando in questo modo il principio di chiusura.

Nel contesto appena introdotto, si vorrebbe estendere il sistema KDDML-MQL con le funzionalità necessarie per la *preparazione dei dati*, intesa come l'insieme di tutte le elaborazioni che precedono la fase di data mining vera e propria.

La preparazione dei dati è spesso la parte più costosa in termini di tempo (si parla del 60% – 80% del tempo complessivo), ed ha un impatto decisivo sui risultati del processo di estrazione di conoscenza, così come sui “costi” della fase specifica di data mining. Nel presente lavoro sono stati raccolti ed analizzati i problemi nella preparazione dei dati al fine di proporre un linguaggio, con il quale affrontare tali problemi. Esso costituisce in questo modo uno strumento dichiarativo in grado di effettuare tutte le trasformazioni necessarie, per adeguare i dati al processo di estrazione di conoscenza.

Le figure a-c descrivono alcune visioni del processo di estrazione di conoscenza dai dati, le quali differiscono solamente nella parte iniziale. Si tratta sempre di schematizzazioni multi-passo del processo, nelle quali l'attività principale è costituita dall'individuazione ed estrazione di modelli di conoscenza (*data mining*), seguita dall'interpretazione e valutazione di tali modelli. Si sono volute riportare diverse visioni del processo KDD, per evidenziare le diverse problematiche che possono presentarsi negli stadi preliminari al data mining. Queste differenze sono dovute alle diverse assunzioni fatte sui dati di partenza; a seconda della provenienza, del formato e della qualità dei dati, possono sorgere diverse problematiche in fase di preparazione dei dati.

Un obiettivo importante nel nostro lavoro è proprio stata la completezza in questo senso, ossia si è cercato di coprire, con il linguaggio per la preparazione dei dati DPL (Data Preparation Language), tutte le più importanti necessità che un utente potesse incontrare nella fase iniziale di un progetto di estrazione di conoscenza. Mentre è già possibile importare nel sistema KDDML dati, archiviati nei più diffusi formati, attraverso un'interfaccia ODBC, è ora auspicabile, sempre all'interno del sistema KDDML, poter integrare dati eterogenei, effettuare la pulizia, e trasformare i dati appositamente per il data mining.

Nel presente lavoro, mettendosi nella situazione più generale possibile, si parte dal solo presupposto che i dati di origine siano in formato elettronico, ma potendo avere origini diverse ed eterogenee: essi possono provenire da semplici basi di dati, da data warehouses o da files, essere strutturati o meno, possono provenire da una singola sorgente, come da sorgenti multiple ed eterogenee. Inoltre essi potrebbero contenere degli errori e delle inconsistenze, o altre forme di anomalie, che devono essere trattate per una buona riuscita del processo di estrazione di conoscenza. Infine, è sempre necessario trasformare i dati in modo da adeguarli alle specifiche degli algoritmi di data mining.

Senza legarci ad una schematizzazione in particolare, tra quelle note in letteratura, né dando un ordine cronologico, sono state individuate le seguenti questioni che possono insorgere nella preparazione dei dati³:

- **Integrazione**

I dati provenienti da fonti eterogenee (DB relazionali, DB deduttive, files di testo, ecc) sono raccolti e combinati, per costruire una visione uniforme degli stessi, a prescindere da quale tipo di sorgente siano stati prelevati.

- **Consolidamento**

Il risultato di questo passo è la creazione di un database, detto **Data Warehouse**, utilizzato per il supporto alle decisioni. Il consolidamento consiste nell'*integrazione* e *pulizia*, ma può comprendere anche passi di *selezione* e *trasformazione*.

- **Preprocessing**

³ Le descrizioni possono essere discordanti con alcune definizioni in letteratura, proprio perché non esiste una terminologia univoca e standardizzata nell'ambito della *preparazione dei dati* alle applicazioni.

I dati strutturati del Data Warehouse vengono preparati per il mining; gli algoritmi di data mining hanno infatti la pretesa di richiedere dati corretti e con un formato ben preciso. L'integrazione prodotta in un momento precedente può aver portato a possibili inconsistenze, ridondanze o errori, che richiedono una sotto-fase di *pulizia*. Ulteriori operazioni che possono ricorrere in questa fase sono la *selezione* e la *trasformazione*. Raramente si considera anche l'*integrazione* come un passo di *preprocessing*.

- **Pulizia**

La pulizia dei dati (*Data Cleaning/Cleansing*) consiste in un processo di miglioramento della qualità del dato attraverso la modifica della forma del dato stesso o del suo contenuto. In questa sede vengono trattati soprattutto gli errori e le inconsistenze:

- si rilevano (e si trattano) i valori scorretti (*errori/rumore*)
- si stimano i valori mancanti
- si trattano gli *outliers* (valori eccezionali, che non sono necessariamente errori)
- si consolidano duplicati e ridondanze
- si risolvono inconsistenze
- si convertono i tipi impropri

- **Selezione/Riduzione**

Questo stadio segmenta e seleziona i dati rilevanti per l'obiettivo di data mining. Esso può ricorrere sia in fase di *consolidamento*, che in fase di *preprocessing*, per selezionare un sottoinsieme di dati su cui una particolare estrazione deve essere effettuata.

La selezione comprende operazioni di *riduzione verticale ed orizzontale*:

- il campionamento,
- l'aggregazione,

- la proiezione.

- **Trasformazione**

Una volta selezionati ed identificati, i dati sono trasformati per essere esplorati dagli algoritmi di data mining. La *trasformazione* può comprendere passi di:

- *selezione* (campionamento, aggregazione, ecc),
- conversione di tipi (valori nominali in numerici, nel caso di successivo utilizzo di reti neurali),
- definizione di attributi nuovi, derivati da elaborazioni sugli altri attributi (applicando operatori matematici o logici),
- riduzione del range del dominio degli attributi (normalizzazione),
- riduzione del numero di possibili valori assunti dai dati (discretizzazione),
- e altre manipolazioni ad hoc per soddisfare le specifiche degli algoritmi.

È importante comunque ricordare che la preparazione dei dati prescinde da un'attenta analisi dei dati. A questo scopo possono servire la documentazione, i tradizionali strumenti di interrogazione delle basi di dati, analisi statistiche e visualizzazione di grafici, ma anche gli stessi modelli di conoscenza estratti da un campione dei dati; tutte funzionalità di cui il sistema KDDML dispone. Durante tale fase preliminare vengono identificati e caratterizzati il dominio e la qualità dei dati, per stabilire fin dall'inizio, se, e fino a che punto, i dati a disposizione possono soddisfare le aspettative.

In questo lavoro non viene studiata una metodologia per la preparazione dei dati, ma piuttosto si identificano le funzionalità indispensabili, e quelle di particolare utilità, che precedono l'estrazione di conoscenza. In particolare, non verrà approfondita la questione dell'analisi dei dati, fornendo ciononostante un utile supporto per la stessa attraverso il meccanismo di *marcatura*, di cui il linguaggio DPL dispone.

Riassumendo, i vincoli posti sullo sviluppo di un linguaggio per la preparazione dei dati sono stati:

- la *completezza* rispetto alle necessità che possono insorgere in tutte le fasi che precedono l'applicazione degli algoritmi di data mining;
- l'*integrazione* del linguaggio, come strumento di preparazione dei dati, in un ambiente già parzialmente sviluppato;
- un approccio formale nella definizione della soluzione proposta, possibilmente *algebrico* ed uniforme a quello adottato per la specifica del linguaggio di interrogazione di data mining, MQL, già integrato nel sistema KDDML-MQL.

I capitoli saranno organizzati come segue:

- **Capitolo 1.** Vengono presentati ed analizzati alcuni strumenti esistenti, in grado di offrire funzionalità utili nell'ambito della preparazione dei dati all'estrazione di conoscenza.
- **Capitolo 2.** Si approfondisce il problema della pulizia dei dati e si fissano i requisiti del sistema, che si considerano necessari per offrire un supporto adeguato alla preparazione dei dati. Successivamente si introduce, ad un alto livello, l'insieme degli operatori proposti per il linguaggio, in particolare attraverso degli esempi di applicazione degli stessi ad alcuni possibili problemi di preparazione. Viene inoltre definita la sintassi del linguaggio, come estensione del linguaggio MQL.
- **Capitolo 3.** Viene fornita la specifica formale del linguaggio attraverso una semantica operativa, conforme a quella adottata nella definizione di MQL: il modello di astrazione adottato per le espressioni del linguaggio è quello delle sequenze, ed i costrutti del linguaggio sono definiti come operatori derivati su di esse.

- **Capitolo 4.** Sono enunciate e dimostrate alcune proprietà algebriche interessanti del linguaggio.
- **Capitolo 5.** Viene presentato un esempio di preparazione dei dati. Si tratta del pre-processamento delle informazioni contenute nei registri di accesso dei web server, al fine di predisporle per una successiva analisi di “clickstream”.
- **Capitolo 6.** Si riassume il lavoro svolto e si forniscono degli spunti per l'estensione del linguaggio e del sistema KDDML, in riferimento alla preparazione dei dati.

Nell'organizzazione della presente tesi si è cercato di rispecchiare l'evoluzione del lavoro svolto. Infatti, si ritiene che un'analisi degli approcci esistenti, aiuti il lettore ad intuire le questioni che emergono durante il processo di preparazione dei dati, oltre che a comprendere più profondamente le soluzioni alle varie problematiche che verranno proposte nel presente lavoro.

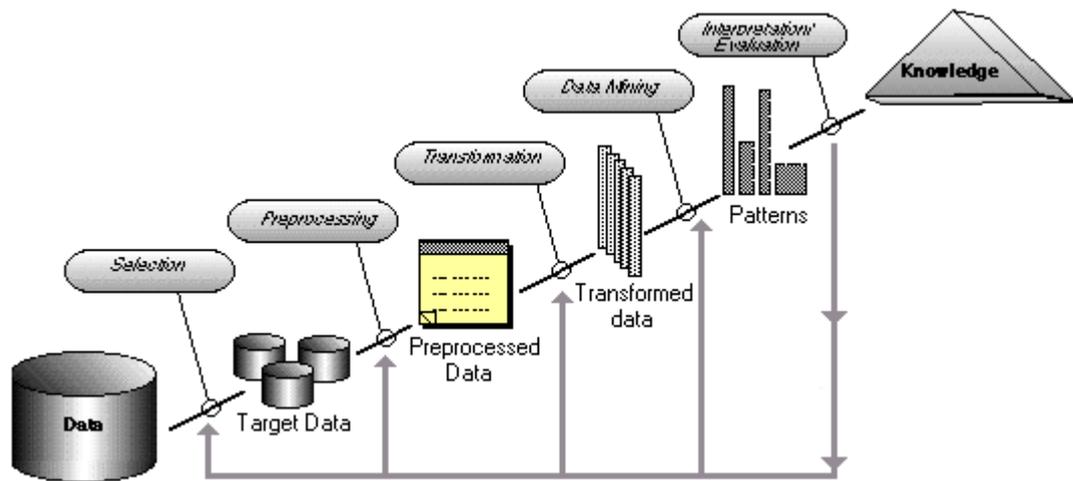


Figura a: Il processo di KDD secondo Fayyad, Piatetsky-Shapiro e Smyth [FPS96]

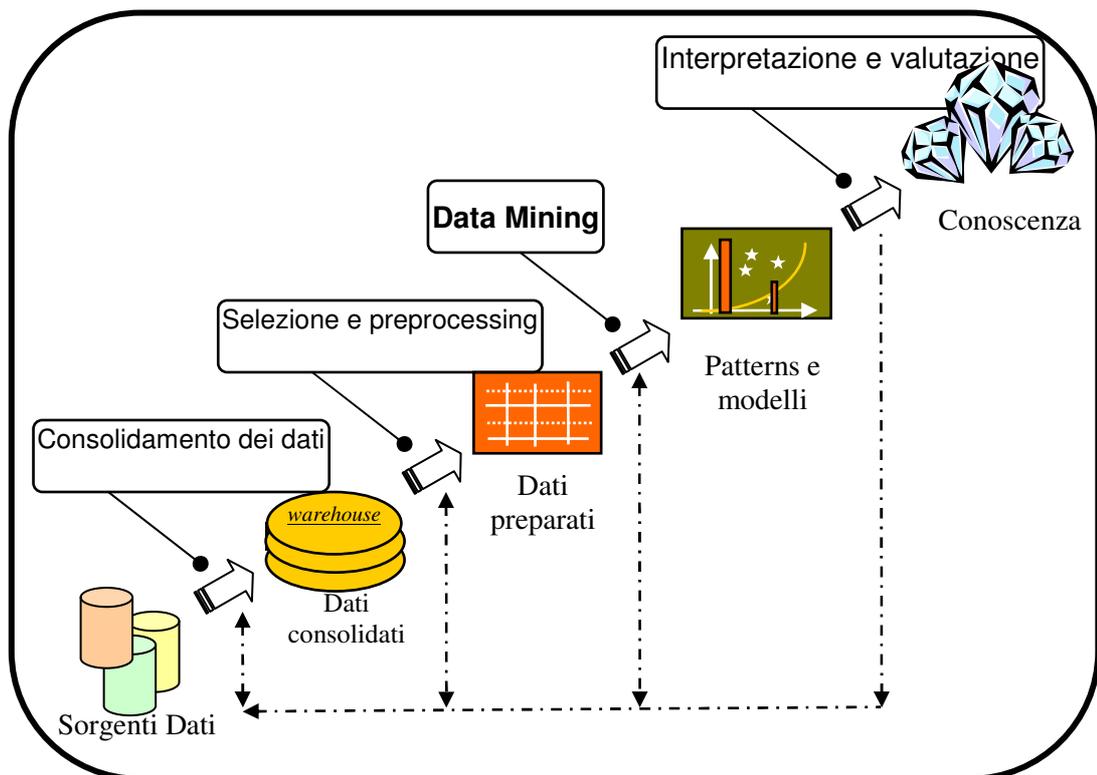


Figura b: Una seconda visione del processo di KDD [Tesi4]

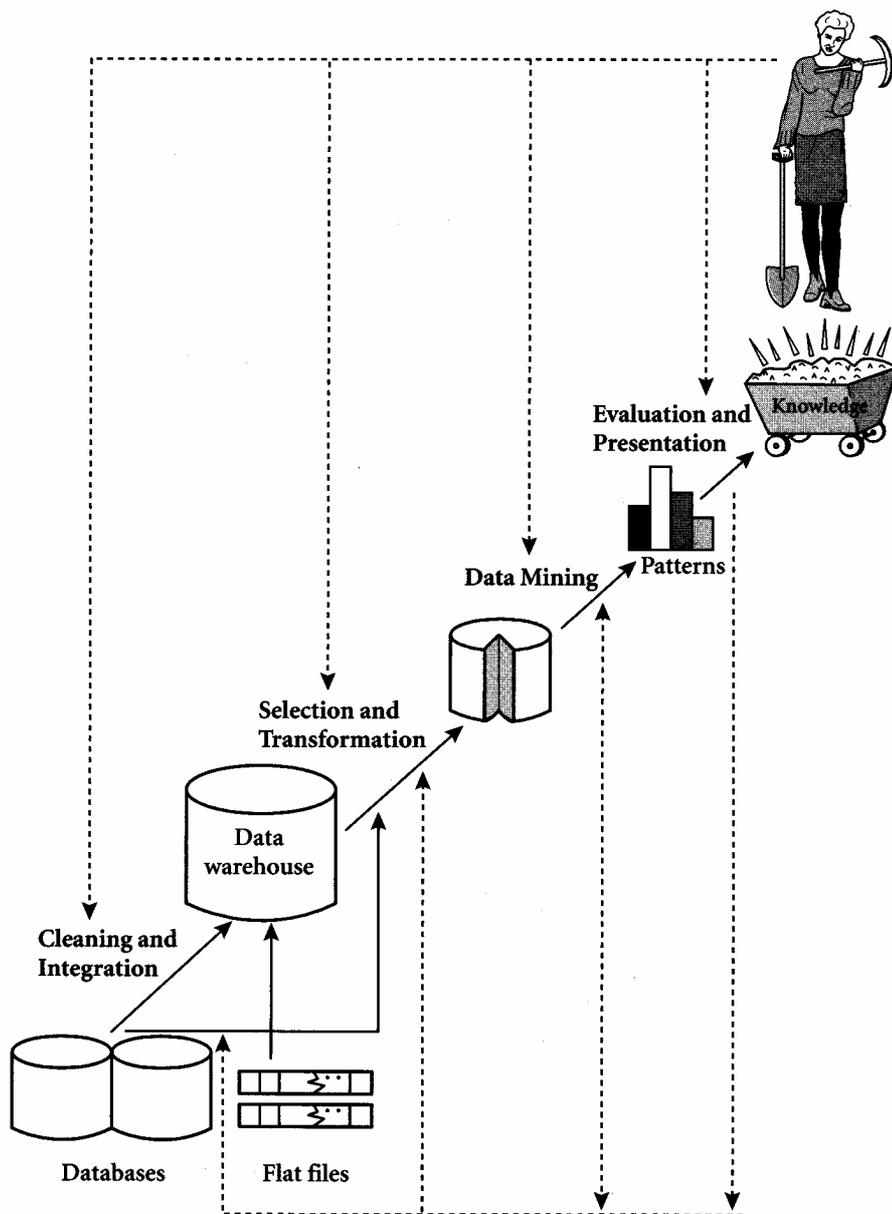


Figura c: Il processo di KDD e dei suoi stadi secondo J. Han e M. Kamber [HK00]

Capitolo 1

Lo stato dell'arte

Nonostante l'importanza che riveste la preparazione dei dati nel contesto di data mining, questo aspetto è inaspettatamente trascurato. Esistono innanzitutto pochi sistemi commerciali che affrontano il problema in modo approfondito, ma la vera lacuna è rappresentata dall'incompletezza di tali strumenti. Infatti, chiunque abbia avuto la necessità di raccogliere e preparare dati all'interno di un progetto di data mining, si è visto costretto ad utilizzare più sistemi per basi di dati, i quali offrono funzionalità diverse, e che si integrano a vicenda. Inoltre è frequente che l'utente, in fase di *data cleaning*, debba ricorrere a linguaggi di programmazione esterni, quali il Perl, per effettuare la "pulizia" dei dati attraverso l'elaborazione delle stringhe, che costituiscono i valori degli attributi in una tabella relazionale. Ed è proprio il cosiddetto *data cleaning*, ossia la "pulizia" dei dati da errori ed inconsistenze, che rappresenta il problema cruciale, e sicuramente anche quello di più difficile trattazione. Esistono anche strumenti che offrono un supporto apposito a tale scopo, e che in letteratura sono noti sotto il nome di *strumenti ETL* (Extraction Transformation Loading). Tuttavia la questione è molto complessa e delicata, e tuttora non si può prescindere, durante la fase di pulizia, ma anche nel complesso della fase di preparazione dei dati, dalle decisioni di un utente esperto. Siamo ancora lontani, infatti, dall'immaginare di poter offrire uno strumento per la preparazione automatica dei dati, e l'esperienza dell'utente rimarrà un elemento indispensabile, anche quando, attraverso il supporto dell'Intelligenza Artificiale, sarà possibile automatizzare parzialmente la fase di preparazione dei dati. Probabilmente è proprio la delicatezza della questione che ha ritardato l'impegno nel campo della ricerca, in cui solo negli ultimi anni, per la forte richiesta di soluzioni alle problematiche di integrazione e di pulizia dei dati, si ha avuto un incremento dell'interesse per la questione.

Durante lo sviluppo del presente lavoro si è preso visione di alcuni tra i sistemi più diffusi e che offrono le funzionalità di *preprocessing* per data mining. Tra questi, in ambito commerciale, si è visto *SQL Server* e *Clementine*, mentre in ambito accademico la scelta è caduta innanzitutto su *WEKA*, una libreria interamente scritta in Java, e che offre funzionalità di data mining e di preprocessing. Inoltre si è preso spunto da altri prodotti nel campo della ricerca, in particolar modo *AJAX*, il quale è stato sviluppato in Francia all'istituto di ricerca INRIA (Institut National de Recherche en Informatique et en Automatique). Esso rappresenta un'estensione al linguaggio SQL, che permette di specificare le trasformazioni all'interno di un processo di *data cleaning*.

Uno degli obiettivi fondamentali del presente lavoro è stato quello di proporre un prodotto completo, che offrisse cioè un supporto durante l'intera fase di preparazione dei dati, senza la necessità di dover ricorrere a strumenti esterni. È a questo scopo che è stato utile analizzare alcuni prodotti esistenti, così come sono stati di notevole ausilio i commenti e le critiche di persone che hanno effettuato la preparazione di dati e che hanno utilizzato tali strumenti. Infine, un ulteriore supporto durante lo sviluppo delle proposte per il formalismo DPL è stato offerto dalla documentazione in rete, costituita principalmente da articoli in cui si affrontano più in dettaglio le molteplici e diverse problematiche della pulizia dei dati.

In seguito verranno analizzati alcuni sistemi che permettono, tra l'altro, di effettuare la preparazione dei dati per data mining, e si cercherà di mettere in evidenza le caratteristiche più rilevanti ed aderenti a questa specifica questione.

1.1 WEKA

WEKA (Waikato Environment for Knowledge Analysis)¹ rappresenta un ambiente *open source*, che offre funzionalità per l'estrazione di conoscenza. Esso è interamente sviluppato in Java, e disponibile sia per Linux che per Windows. È stato sviluppato all'Università di Waikato, in Nuova Zelanda, ed in campo accademico è probabilmente il prodotto più conosciuto ed anche completo. Infatti l'ambiente WEKA, non solo

¹ [WEKA]

possiede la più ricca libreria di algoritmi di data mining, ma dispone anche di funzionalità di preprocessing, e di analisi e visualizzazione dei risultati.

Attualmente WEKA costituisce già una componente importante del sistema KDDML, e fornisce le funzionalità proprie della fase di data mining. In futuro, all'interno dell'implementazione del linguaggio di preprocessing DPL, questo prodotto potrebbe fornire un ulteriore supporto nella fase di preparazione dei dati. In particolare, il package *weka.filters* raccoglie tutte le classi che implementano i vari tipi di filtri, i quali permettono di trasformare i dati prima di effettuare l'estrazione di conoscenza. I pacchetti *weka.associations*, *weka.clusterers* e *weka.classifiers* contengono le classi che implementano gli algoritmi di data mining, mentre il pacchetto *weka.gui* implementa l'interfaccia grafica dell'ambiente. Infine, *weka.core* rappresenta il pacchetto centrale, nel quale sono implementate le classi utilizzate da tutti gli altri pacchetti.

Una caratteristica di WEKA è il fatto che costituisce un ambiente aperto, di cui l'utente può utilizzare solo le componenti che rispondono alle proprie richieste. Si noti infatti che è possibile utilizzare questo strumento secondo tre modalità diverse:

1. si possono richiamare dalla riga di comando gli algoritmi della libreria di WEKA, implementati in Java, per effettuare trasformazioni ed estrazione di conoscenza dalle proprie collezioni di dati;
2. è possibile utilizzare esclusivamente le funzionalità offerte, interagendo con il sistema attraverso un'interfaccia grafica apposita, messa a disposizione da WEKA;
3. si può infine utilizzare solo alcune componenti della libreria, come moduli da inserire all'interno di applicazioni proprie.

Si noti che, in ogni caso, l'utilizzo di WEKA comporta una limitazione nel formato dei dati. Essi infatti devono essere contenuti in files nel formato proprietario ARFF. Tuttavia non si tratta che di una variazione al diffuso formato CSV (Comma Separated Values). Essi sono infatti files di testo ASCII in cui le tuple sono separate da un carattere di *caporiga*, e i valori all'interno di una singola tupla sono separati da una virgola. Nei files ARFF è prevista anche un'intestazione che contiene il nome ed il tipo degli attributi; informazioni che sono indispensabili per applicare gli algoritmi di data

mining. Gli attributi in WEKA possono assumere cinque diversi tipi: *reale*, *intero*, *enumerazione*, *stringa* e *data*.

Un file ARFF inizia dunque con una riga in cui si dichiara il nome della relazione, attraverso la specifica *@relation nome*. Le righe successive sono del tipo *@attribute nome tipo*, ed indicano il nome ed il tipo degli attributi che compongono la relazione. Infine seguono i dati che costituiscono il contenuto della base di dati, i quali sono preceduti dalla parola chiave *@data*.

Un esempio chiarificante è dato in figura 1.1.

<i>Prospettiva</i>	<i>Temperatura</i>	<i>Umidità</i>	<i>Vento</i>	<i>PlayTennis</i>
Sole	85	85	Debole	Yes
Sole	80	90	Forte	No
Coperto	83	86	Forte	Yes
Pioggia	70	96	Debole	No

```

@relation PlayTennis

@attribute Prospettiva {Sole, Coperto, Pioggia}
@attribute Temperatura real
@attribute Umidità real
@attribute Vento {Forte, Debole}
@attribute PlayTennis {Yes, No}

@data
Sole, 85, 85, Debole, No
Sole, 80, 90, Forte, No
Coperto, 83, 86, Forte, Yes
Pioggia, 70, 96, Debole, Yes

```

Figura 1.1: La tabella ed il file in formato ARFF corrispondenti alla relazione *PlayTennis*

Come anticipato, il formato ARFF rappresenta in realtà una semplice estensione al noto formato CSV, in cui molti sistemi per la gestione di dati consentono già di salvare i dati. La successiva trasformazione da file CSV ad ARFF è immediata, e quindi non si ritiene limitante questa restrizione.

Una critica può essere fatta all'interfaccia grafica per interagire con il sistema, la quale risulta un po' spartana: colori monotoni, niente effetti speciali. Essa richiede inoltre una approfondita conoscenza nel settore da parte degli utenti. Tuttavia bisogna considerare che WEKA non è stato sviluppato per scopi commerciali, e non è indirizzato ad un pubblico vasto. Essa è rivolta invece, in ambito accademico, ad esperti, come strumento di supporto all'estrazione di conoscenza. Si ritiene perciò che anche questo non sia un limite importante sul quale soffermarsi. La figura 1.2 mostra l'aspetto dell'interfaccia grafica durante la specifica di una classificazione bayesiana.

Il vanto principale di WEKA consiste probabilmente nell'essere un sistema aperto, e continuamente in sviluppo, che si lascia facilmente integrare in qualunque ambiente in cui sia richiesta qualche funzionalità di estrazione di conoscenza. Inoltre, essendo la libreria WEKA fornita sia di un pacchetto per il preprocessing, che di pacchetti per la fase di data mining, essa offre la possibilità di integrare ed alternare le due suddette fasi, all'interno di un unico ambiente. Questo rappresenta un aspetto molto interessante, ed un requisito che ci si impone anche nello sviluppo del sistema KDDML. Si è infatti riconosciuto come, anche all'interno della fase di preparazione dei dati, notevoli vantaggi possono essere ottenuti tramite gli algoritmi di data mining. È intuitivo, ad esempio, che il problema dei dati assenti possa essere affrontato attraverso l'applicazione di un "predittore". Un ulteriore esempio è costituito dalla ricerca di *outliers*, ossia valori "fuori norma", tramite l'applicazione di un classificatore costruito su di un *test set* privo di eccezioni. Si ritiene perciò che un punto di forza di WEKA sia rappresentato dal fatto che esso offre, in un unico ambiente, funzionalità di data mining, ma anche di preprocessing e di visualizzazione dei risultati.

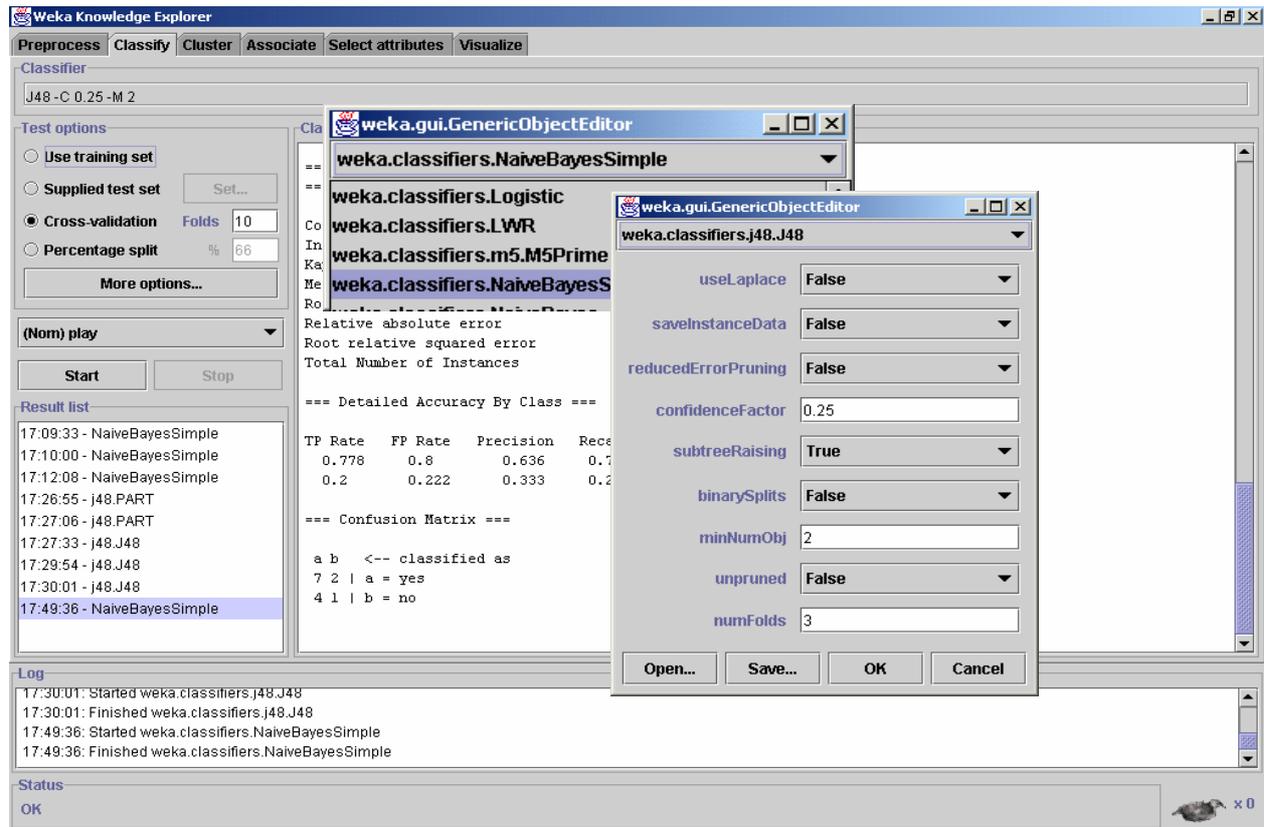


Figura 1.2 : Interfaccia grafica di WEKA

Una lacuna del prodotto, in seguito ad un'analisi globale della problematica di preparazione dei dati, è risultata essere la mancanza delle tipiche elaborazioni sulle stringhe. Sono infatti molto frequenti, in fase di pulizia dei dati, le trasformazioni dei valori di tipo stringa. Per trattare questo problema è allora necessario ricorrere a script, tipicamente in Perl, in cui è possibile specificare lo schema di costruzione di una generica stringa, ed in cui è implementato il meccanismo di *pattern matching*. Solo con l'ultima versione della piattaforma *Java 2 Platform – Standard Environment (J2SE)*, è fornito un supporto più potente per l'elaborazione delle stringhe. Essa infatti è dotata del pacchetto *java.util.regex*, il quale contiene tutte le classi che realizzano il meccanismo di pattern matching. È allora possibile definire un *pattern*, ossia uno schema di costruzione di una stringa, attraverso un'espressione regolare. Le espressioni regolari sono volutamente molto simili a quelle del linguaggio Perl, e di altri linguaggi di scripting. Un oggetto della classe *Pattern*, associato ad una particolare espressione regolare, rappresenta allora una classe di stringhe, le quali rispettano il pattern

specificato. Se si vuole effettuare il pattern matching, tra tale pattern ed un valore di tipo stringa, alla stringa viene associato un oggetto della classe *Matcher*, ed è possibile effettuare diversi tipi di matching attraverso i molteplici metodi di questa classe.

È lecita una considerazione critica alla documentazione. Questa è costituita prevalentemente dalla documentazione web, *Javadoc*, generata in modo automatico. Per l'utente che intende utilizzare il sistema esclusivamente attraverso l'interfaccia grafica esiste un tutorial, piuttosto tecnico, costituito dal capitolo 8 del libro *WEKA Machine Learning Algorithms in Java*. Tuttavia chi è interessato ad utilizzare la libreria, oppure singoli pacchetti, da integrare come moduli all'interno di un sistema proprio, si troverà di fronte a qualche difficoltà. L'unica documentazione sulle classi ed i metodi della libreria è costituita dalla documentazione API, generata in modo automatico. Anche gli esperti del settore devono generalmente perdere un po' di tempo dietro a tentativi, guidati dalla conoscenza e dal buon senso.

Infine, si osserva che non è stata posta molta attenzione all'efficienza del sistema, né alla scalabilità: ogni dataset è mantenuto in memoria. Ma questi non rappresentano requisiti indispensabili per un sistema nato per scopi di ricerca.

Complessivamente si deve comunque dare un ottimo giudizio al prodotto sviluppato all'Università di Waikato, per i suoi pregi nominati sopra. La documentazione scarna dei pacchetti potrebbe essere una limitazione voluta, in quanto protezione dall'utilizzo di componenti singole all'interno di sistemi esterni. È testimone della qualità di WEKA il fatto che, ciononostante, questo prodotto è il più noto e diffuso nei progetti in ambito accademico.

1.2 Clementine

Clementine è un prodotto commerciale SPSS [Clem]. Esso è largamente riconosciuto come il primo ambiente di data mining, che abbia realmente supportato l'intero processo di estrazione di conoscenza.

Come Weka, Clementine dispone di un ambiente grafico (vedi fig. 1.3), che è molto curato, a differenza del primo. Esso offre delle metafore di interazione visuale, per facilitare la comprensione e l'utilizzo dello strumento. Infatti il processo di estrazione di

conoscenza è visto come una sequenza di passi di trasformazione del database, e la metafora dell'ambiente grafico è rappresentata da un diagramma di flusso. In tale diagramma, i nodi rappresentano i dati, ad esempio possono corrispondere ad una relazione di una base di dati, ad un file di testo, oppure ad una vista relazionale, e gli archi indicano le elaborazioni che i dati subiscono. Complessivamente, il processo di estrazione di conoscenza dai dati può essere dunque visto come un flusso, che parte, tipicamente, da nodi associati a tabelle relazionali di un database, e termina in un nodo che rappresenta un modello di conoscenza. L'utente può definire un flusso, il quale viene visualizzato sul desktop, la finestra principale dell'ambiente grafico.

Innanzitutto è necessario specificare i nodi di partenza, che devono essere inizializzati con i dati da elaborare. Come sorgenti dei dati si possono utilizzare, ad esempio, basi di dati esterne, attraverso una connessione secondo la specifica comune di accesso ai dati ODBC (Open Database Connectivity). Altre sorgenti possono essere un file di testo in formato CSV (Comma Separated Values), un file EXCEL, o altri ancora. Il diagramma di flusso viene costruito progressivamente, inserendo nuovi nodi, e questi vengono collegati al grafo parziale attraverso degli archi. Questi ultimi sono definiti specificando il tipo di elaborazione, ed i parametri necessari, che i dati nel nodo di partenza subiscono. Una "paletta di oggetti" è a disposizione dell'utente durante la fase di costruzione del diagramma (vedi fig. 1.3). È anche possibile definire un "super-nodo" formato da un sottografo, ed utilizzarlo come componente *nodo* nella costruzione di un diagramma di flusso. In questo modo si può dare una struttura a livelli al processo di estrazione di conoscenza, il cui diagramma potrebbe altrimenti raggiungere dimensioni eccessive. Inoltre, un *super-nodo* serve anche a definire una sequenza frequente di elaborazioni, e realizza in questo modo l'astrazione funzionale su un linguaggio grafico di programmazione.

Una volta definito il diagramma di flusso, si può mandare in esecuzione il processo, al termine del quale saranno stati elaborati i dati associati ai nodi intermedi, ed estratti i modelli di conoscenza associati ai nodi terminali. Una finestra laterale consente di visualizzare tutti i modelli generati. A questo punto è possibile utilizzarli per il loro scopo ultimo, ossia come supporto alle decisioni. Ma non si ferma qui l'utilità della conoscenza estratta da una collezione di dati. Un modello può infatti essere riutilizzato sia nella fase di preprocessing, che nella fase di data mining. Ed è proprio per questo che è interessante offrire un unico ambiente per tutte le fasi del processo di estrazione di

conoscenza, che va dal preprocessing fino alla rappresentazione e all'applicazione dei modelli estratti.

Clementine offre inoltre strumenti di analisi statistica e di visualizzazione dei dati, così come metafore di rappresentazione dei modelli di conoscenza.

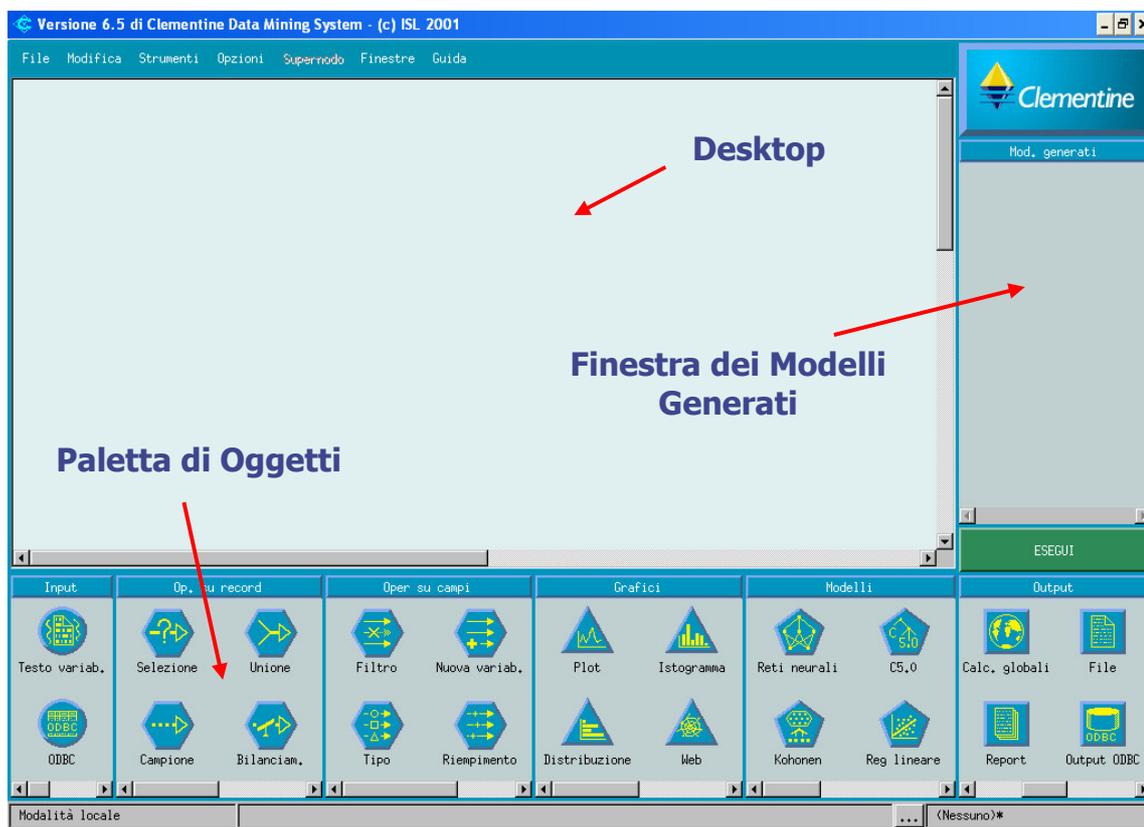


Figura 1.3: Ambiente grafico di Clementine: il desktop e la paletta degli oggetti

Nella fase di preparazione dei dati, in generale, è frequente la necessità di esportare i dati per effettuare il preprocessing con strumenti esterni al sistema. Ad esempio, si preparano i dati all'interno di sistemi per basi di dati, come SQL Server, Access, ma anche in Excel, oppure a livello più basso ancora, attraverso l'utilizzo di linguaggi di programmazione, come il C o il Perl. Una volta effettuate le operazioni di preprocessing in maniera esterna, bisogna importare i dati nel sistema. Clementine, da questo punto di vista, è probabilmente il sistema più completo. Ciononostante si è notato, che le funzionalità di preprocessing di Clementine sono piuttosto specifiche per il data mining. Sono disponibili le operazioni per adeguare i dati ai requisiti dei più importanti

algoritmi di data mining. Non mancano le trasformazioni di riduzione orizzontale dei dati, come anche di riduzione verticale, quali il campionamento ed l'aggregazione. È possibile discretizzare, bilanciare e normalizzare i dati, trasformare il tipo dei dati, selezionarli con operazioni simili ad SQL. Tuttavia si prescinde da un data warehouse piuttosto pulito e ben integrato. Esiste da un lato la possibilità di ricercare e trattare valori mancanti ed outliers, ma c'è una lacuna per quanto riguarda le problematiche di *data cleaning* tipiche dei dati provenienti da sorgenti diverse. Infatti nessun supporto è fornito per l'integrazione di formati diversi, per la rilevazione di duplicati e per l'elaborazione di stringhe. Se l'utente dovesse averne la necessità, deve ricorrere a qualche sistema per basi di dati, come ad esempio SQL Server, che offre funzionalità di preprocessing più generiche. Clementine focalizza indubbiamente sui compiti di data mining, ed il preprocessing è mirato a tale scopo.

Come si è detto sopra, il processo di estrazione di conoscenza viene specificato attraverso un diagramma di flusso. Per quanto riguarda la preparazione dei dati, le operazioni di preprocessing si lasciano facilmente integrare in qualunque punto, come sottografo del processo globale di estrazione di conoscenza. Proprio questa caratteristica è un punto di forza di Clementine, e da cui è stato preso spunto durante lo sviluppo del nostro lavoro; lo stesso obiettivo si è infatti potuto raggiungere, proponendo il linguaggio di preprocessing DPL come un'estensione del linguaggio di interrogazione di data mining MQL, già completamente integrato nel sistema KDDML.

Per concludere, si riporta la nota di un utente, il quale riferisce che, anche in piccoli progetti, la fase di preparazione dei dati, in Clementine, può richiedere la definizione di diagrammi di flusso di grandi dimensioni. Infatti le operazioni di preprocessing costituiscono spesso la gran parte del diagramma, e questo, perché è necessario introdurre un nodo per ogni singola elaborazione, anche se banale. Anche un singolo comando di SQL richiede molti nodi di elaborazioni. Purtroppo questa è una caratteristica intrinseca del processo di preparazione dei dati, e a tal proposito, si ritiene che la possibilità di definire dei *super-nodi* rappresenti un'ottima risposta al problema. Tale osservazione è stata presa in considerazione anche nei confronti del formalismo DPL, e la soluzione che si propone è conforme a quella di Clementine: un meccanismo di astrazione funzionale permetterebbe di definire e memorizzare una sequenza frequente di operazioni di preprocessing, e di limitare la lunghezza dei programmi.

1.3 SQL Server

SQL Server è una piattaforma commerciale della Microsoft [SQLServer], per la gestione di basi di dati. Si tratta di un sistema molto potente, che, oltre all'amministrazione e l'interrogazione di basi di dati centralizzate, offre strumenti per gestire basi di dati parallele e distribuite. Inoltre esso dispone di strumenti OLAP (On-Line Analytical Processing) e di data warehousing, e recentemente anche di qualche funzionalità di data mining.

Il frammento OLAP è la sorgente di informazioni più largamente utilizzata per le diverse attività di mining. Esso consente agli utenti la visione di dati aggregati attraverso molte dimensioni e gerarchie, così come il calcolo e la trasformazione di dati lungo tali dimensioni e gerarchie, la navigazione all'interno di esse e la visione di sottoinsiemi. OLAP fornisce in questo modo un modello concettuale adeguato per sistemi di supporto alle decisioni, ed un ambiente warehouse fornisce il livello fisico per tale modello concettuale.

Il modello concettuale più popolare è senz'altro costituito dalla visione *multidimensionale* dei dati, dove gli oggetti sotto analisi sono un certo numero di *misure numeriche*. Ogni misura numerica dipende da un insieme di *dimensioni*, che forniscono il contesto della misura. Per esempio le dimensioni associate agli oggetti di una relazione *totale vendite* potrebbero essere l'area geografica, il nome del prodotto, e la data in cui la spesa è stata effettuata. Tutte tre le dimensioni insieme definiscono la misura di un oggetto. All'interno di questo modello concettuale, la metafora di rappresentazione di un oggetto è il *cubo*, che è una struttura multidimensionale contenente dimensioni e misure. Le dimensioni definiscono la struttura del cubo, mentre le misure specificano i valori numerici che definiscono l'istanza di un oggetto. Le posizioni delle celle all'interno del cubo sono definite dall'intersezione dei membri delle dimensioni, mentre i valori delle celle risultano dall'aggregazione dei valori delle misure. Come anticipato, i cubi vengono utilizzati in operazioni di elaborazione analitica in linea (OLAP), e rendono disponibile un meccanismo di facile utilizzo per l'esecuzione di interrogazioni sui dati, con tempi di risposta rapidi e omogenei.

In Microsoft SQL Server 2000 è stato esteso e rinominato il precedente componente *OLAP Services*, ora chiamato *Analysis Services*. Le funzionalità di analisi di OLAP Services, introdotto in SQL Server versione 7.0, sono state potenziate con numerose funzioni nuove e migliorate. Una delle funzioni del nuovo Analysis Services è il data mining, utilizzabile per individuare informazioni in cubi OLAP e database relazionali.

La nostra attenzione è stata rivolta principalmente alle possibilità offerte per operare la preparazione dei dati. Rispetto ai dati trattati, SQL Server permette di importare i più diffusi formati di dati, files di testo ASCII, inclusi i files in formato CSV (Comma Separated Values), così come fogli di calcolo di Excel. Si possono importare basi di dati esterne attraverso delle origini dati ODBC (ad esempio, database Oracle), oppure origini dati OLE DB (ad esempio, altre istanze di SQL Server).

ODBC (Open Database Connectivity) è un'interfaccia a livello di chiamata (CLI) che consente alle applicazioni C e C++ di accedere ai dati. Le origini dati ODBC includono dati archiviati in formati diversi, e attraverso un driver ODBC, SQL Server consente di accedervi. ODBC è stato ampiamente adottato dai programmatori di database, e molti fornitori di database o di terze parti offrono driver ODBC.

Anche OLE DB è una specifica comune di accesso ai dati, ma definita da Microsoft. In molti programmi per l'archiviazione dei dati, quali fogli di calcolo, database o altre applicazioni server, sono disponibili provider OLE DB, che consentono a un'applicazione di accedere ai dati. In OLE DB 2.5 sono incluse inoltre estensioni multidimensionali che consentono ai provider OLE DB di esporre anche le informazioni dei cubi multidimensionali.

Pertanto SQL Server consente di stabilire connessioni a dati archiviati in un'ampia gamma di formati compatibili con OLE DB oppure ODBC.

Una volta a disposizione i dati, essi possono essere trattati. Ovviamente sono disponibili tutte le operazioni di selezione tramite SQL e di modifica del tipo dei dati, così come molteplici ed interessanti funzioni di analisi statistica e di visualizzazione dei dati. In particolare, rispetto a Clementine, esso offre migliori strumenti per l'integrazione di dati eterogenei. Tuttavia, non essendo SQL Server un ambiente per l'estrazione di conoscenza, le operazioni di preprocessing sono piuttosto generali, e mancano le più importanti trasformazioni richieste dagli algoritmi di data mining. Il motivo, per cui è stato preso in considerazione SQL Server nel presente lavoro, è dovuto al fatto che questo strumento è utilizzato molto frequentemente da chi effettua la

preparazione dei dati per data mining. Come si è osservato già nel paragrafo precedente, nei confronti di Clementine, gli ambienti per l'estrazione di conoscenza offrono funzionalità di preprocessing piuttosto specifiche, ad hoc per gli algoritmi di cui il sistema dispone. Invece per quanto riguarda l'integrazione dei dati è necessario ricorrere a sistemi di gestione per basi di dati. La problematica di pulizia è ancora un'altra questione, affrontata con strumenti appositi, noti in letteratura come *ETL tools* (Extraction Transformation Loading tools), oppure effettuata direttamente a basso livello con linguaggi di programmazione, quali il C o il Perl. Un obiettivo importante nel nostro lavoro è proprio stato la completezza in questo senso, ossia si è cercato di coprire, con il linguaggio di preprocessing DPL, tutte le più importanti necessità che un utente potesse incontrare nella fase iniziale di un progetto di estrazione di conoscenza. Mentre è già possibile importare nel sistema KDDML dati, archiviati nei più diffusi formati, attraverso un'interfaccia ODBC, è ora auspicabile, sempre all'interno del sistema KDDML, poter integrare dati eterogenei, effettuare la pulizia, e trasformare i dati appositamente per il data mining.

1.4 AJAX

AJAX è un prodotto sviluppato all'istituto di ricerca INRIA (Institut National de Recherche en Informatique et en Automatique) [GFS00] [GFSS00], il quale permette la specificazione e l'esecuzione di programmi di *pulizia dei dati*. Si tratta di un'estensione di SQL, attraverso cui l'utente può effettuare il *data cleaning* in modo dichiarativo.

In AJAX, un processo di pulizia dei dati accetta in ingresso un insieme di flussi di dati, i quali possono contenere errori ed inconsistenze. Tali flussi possono essere costituiti da files, contenenti record di lunghezza fissa, oppure da tabelle relazionali. Il processo restituisce alla fine dei flussi di dati consistenti, formattati e privi di errori. Un'assunzione di base è che all'interno di ogni singolo flusso i records siano identificati in modo unico tramite una chiave.

Il processo di pulizia è quindi modellato come un grafo, aciclico ed orientato, di trasformazioni, che partono da qualche sorgente di dati e restituiscono dei dati "puliti". Tali trasformazioni sono dette *atomiche*, e sono specificate attraverso dei comandi SQL

estesi con un insieme di primitive specifiche. Ogni trasformazione atomica accetta in ingresso uno o più flussi di dati, e restituisce a sua volta uno o più flussi di dati, i quali possono costituire l'ingresso per trasformazioni successive.

Sono previsti quattro tipi di trasformazioni atomiche:

1. le trasformazioni di *mapping*, le quali “standardizzano”, quando possibile, il formato dei dati, oppure semplicemente producono records nel formato più opportuno;
2. le trasformazioni di *matching*, per individuare le coppie di records che probabilmente riferiscono lo stesso oggetto reale. I records sono comparati sui valori di uno o più campi, usando dei criteri di confronto che possono essere arbitrariamente complessi. Ad ogni coppia di records messi a confronto viene associato un valore di “similitudine”, rappresentato dal risultato della valutazione del criterio.
3. Le trasformazioni di *clustering* raggruppano le coppie di records che hanno un alto grado di “similitudine”, applicando un particolare criterio di raggruppamento, come ad esempio la chiusura transitiva.
4. Infine, le trasformazioni di *merging* fondono i records di ogni cluster in un'unica tupla, formando una nuova sorgente di dati.

Si possono considerare come trasformazioni atomiche anche le viste SQL, tramite le quali si effettuano le tradizionali unioni, le giunzioni, le restrizioni e le proiezioni.

Ad un livello superiore sono definite altre trasformazioni, dette *trasformazioni complesse*. Ogni trasformazione complessa è composta da una sequenza di una o più trasformazioni atomiche, e svolge un compito composto. Sono definite tre trasformazioni complesse:

- la *formattazione*, il cui scopo è di estrarre la struttura dai dati, o di convertire i formati;
- l'*eliminazione dei duplicati*, attraverso cui si ricercano ed eliminano i duplicati “approssimati” all'interno di un flusso di dati;
- il *matching tra tabelle multiple*, che effettua un join tra flussi distinti, basato sulla “similitudine”, e consolida il flusso così ottenuto.

In figura 1.4 sono riassunti i tipi di trasformazione previsti in AJAX, ed è riportato un esempio della loro applicazione all'interno di un processo di pulizia dei dati. In particolare, il programma si riferisce all'esempio *Telecom* riportato nel paragrafo 2.5.4.

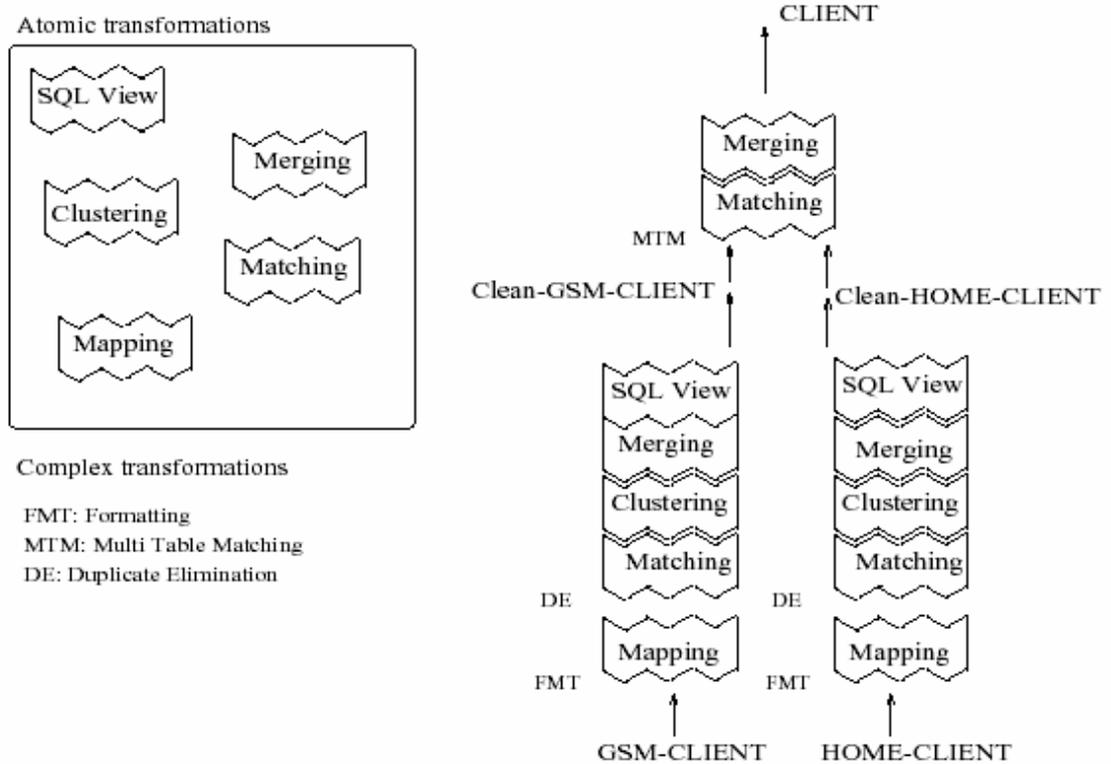


Figura 1.4: I tipi di trasformazione di AJAX ed un processo di pulizia di dati riferito all'esempio *Telecom*

Di seguito è riportato l'esempio di una specificazione di trasformazione atomica di *matching*. I due flussi in ingresso sono costituiti dalla stessa tabella relazionale GSM-CLIENT, e il flusso prodotto in uscita, nominato M1, contiene tutte le (chiavi delle) coppie di records, il cui grado di similitudine è stato valutato superiore a 0,5.

Il sistema dispone di una libreria predefinita di funzioni esterne, scritte in un linguaggio di programmazione, che l'utente può estendere secondo le proprie esigenze. Tra queste ci sono diverse funzioni per calcolare la "distanza" tra due stringhe, come ad esempio la distanza di Hamming. La similitudine è calcolata in questo esempio da una funzione esterna *nameSimF()*.

```
CREATE MATCHING M1  
FROM GSM-CLIENT g1, GSM-CLIENT g2  
LET similarity = nameSimF(g1.name, g2.name)  
WHERE g1.gsmlid < g2. gsmlid  
AND similarity > 0.5  
{ SELECT g1. gsmlid AS gsmlid1, g2. gsmlid AS gsmlid2, similarity AS similarity  
KEY gsmlid1, gsmlid2 }
```

La natura dichiarativa del linguaggio è ereditata direttamente da SQL. Tuttavia il linguaggio di specificazione di AJAX non è completamente dichiarativo, in quanto può contenere frammenti di codice imperativo, in modo da essere adattato alle richieste di una particolare applicazione di *cleaning*. Infatti il linguaggio può essere esteso nei seguenti modi, permettendo di esprimere criteri di cleaning in maniera potente:

- le trasformazioni possono invocare funzioni esterne, su un dominio specifico (ad esempio funzioni di normalizzazione o di matching), e queste possono essere aggiunte ad una libreria predefinita di funzioni;
- le trasformazioni possono essere combinate con dei tradizionali comandi SQL in modo da ottenere anche complessi programmi di pulizia dei dati;
- l'insieme di algoritmi proposti per effettuare il clustering può essere esteso a piacere.

Inoltre, il linguaggio di specificazione di AJAX consente che l'utente sia coinvolto in maniera esplicita durante l'esecuzione di un programma, sia per trattare i casi eccezionali che si verificano durante l'esecuzione della pulizia, che per ispezionare i risultati intermedi.

Infine AJAX è l'unico sistema che supporta un meccanismo di *tracciabilità*. L'utente può effettuare il backtracking di ogni trasformazione all'interno di un programma di pulizia dei dati, in modo da scoprire i record di ingresso che hanno generato un certo record in uscita. Le informazioni ottenute con il backtracking possono servire per raffinare i criteri di pulizia ed iterare il processo. Per supportare questo meccanismo, AJAX memorizza annotazioni sui metadati durante l'esecuzione di un programma.

È da notare che, in AJAX si è cercato di sfruttare quanto più possibile le funzionalità fornite dai DBMS: il linguaggio, l'esecuzione e l'ottimizzazione. In

secondo luogo, AJAX non fornisce nessun meccanismo per aiutare l'utente nella decisione dei criteri da applicare durante l'integrazione e la pulizia. Questi criteri sono fortemente legati all'applicazione ed al dominio dei dati, e sta all'utente esperto del settore fissarli. Egli conosce il dominio d'applicazione, ed è posto quotidianamente di fronte ad anomalie nei dati. AJAX mette solamente a disposizione un insieme di primitive di basso livello, che l'utente può utilizzare sfruttando l'esperienza accumulata.

Un prossimo obiettivo consiste nel fornire un nuovo insieme di trasformazioni atomiche, in cui sono integrati metodi dal campo della Statistica e dell'Apprendimento Automatico, e che forniscono un supporto nell'analisi dei dati ed aiutano a rilevare i problemi di dati "sporchi".

Si vuole infine fare notare come è stato affrontato in AJAX un problema importante: quello di situazioni anomale che possono verificarsi durante il processo di pulizia. Infatti, proprio perché i dati sono "sporchi", sia per quanto riguarda il formato che il contenuto, possono verificarsi le situazioni più imprevedibili. AJAX consente di gestire le eccezioni che vengono sollevate durante le operazioni di cleaning, distinguendo tra *eccezioni anticipate* ed *eccezioni non anticipate*. Al verificarsi di queste, i records del flusso che le hanno generate vengono "marcati", e possono successivamente essere rintracciati e trattati opportunamente. Le eccezioni anticipate sono quelle previste nel programma, ed è possibile così marcare i records secondo le esigenze specifiche dell'applicazione. Invece le eccezioni non anticipate sono tutte quelle che sono previste e gestite in modo implicito da ogni trasformazione.

Si considera che questo approccio sia un'ottima soluzione per affrontare il problema delle situazioni anomale durante la pulizia dei dati. In questo modo si consente contemporaneamente una certa interattività dell'utente, senza richiedere necessariamente di interrompere l'esecuzione delle trasformazioni. Infatti, la mole di dati nel contesto di estrazione di conoscenza può raggiungere dimensioni tali, che può essere preferibile non dover sospendere il processo ad ogni eccezione, pur tenendo traccia di tutte le anomalie riscontrate.

In conclusione, di AJAX si apprezza l'estendibilità, un obiettivo comune al sistema KDDML. Inoltre, la natura dichiarativa del suo linguaggio di specificazione, lo rende il prodotto che maggiormente si avvicina all'approccio adottato durante lo sviluppo di DPL. Per questo motivo è stato considerato un interessante punto di riferimento e di confronto.

AJAX rimane comunque uno strumento specifico per *data cleaning*, pertanto non dispone di primitive apposite per la preparazione dei dati all'estrazione di conoscenza. Si noti ciononostante che, attraverso la libreria di funzioni esterne, è possibile estendere il sistema con algoritmi ad hoc per specifiche applicazioni. Rimarrebbe però sempre impossibile effettuare la preparazione dei dati come un processo iterativo che comprende fasi di estrazione di conoscenza. Proprio questa possibilità si è rivelata infatti molto interessante e promettente, e nell'ambiente per l'estrazione di conoscenza KDDML è possibile realizzarla.

1.5 Considerazioni finali

Nei paragrafi precedenti si sono analizzati i più noti strumenti di preprocessing, in riferimento al contesto di data mining. Si è cercato, a tale proposito, di evidenziare le caratteristiche principali degli strumenti, assieme a quelle ritenute più interessanti.

Innanzitutto è importante notare come ognuno dei prodotti descritti difficilmente riesca a supportare tutta la preparazione dei dati per scopi di data mining. Prendendo spunto proprio dalle lacune dei singoli sistemi, così come dalle funzionalità specifiche offerte, si è considerato il problema della preparazione dei dati come costituito da tre questioni: (i) l'integrazione ed il consolidamento nella struttura, di dati provenienti da sorgenti diverse, (ii) il consolidamento dei dati nel contenuto, ed infine (iii) le trasformazioni ad hoc per il data mining, in modo da adeguare i dati per l'applicazione degli algoritmi. Come si è potuto constatare, queste ultime trasformazioni sono presenti negli ambienti per data mining, come ad esempio Clementine o WEKA, mentre sono del tutto assenti nei sistemi per basi di dati e negli strumenti di pulizia (ETL). Per quanto riguarda l'integrazione di sorgenti di dati multiple, ormai tutti i sistemi consentono di formattare i dati, e di effettuare trasformazioni di schema. Invece il consolidamento nel contenuto, di dati provenienti da sorgenti diverse, è un problema importante e delicato, che tuttora lascia delle questioni irrisolte. A questa classe appartengono: il consolidamento del formato di dati non strutturati, la rilevazione ed il consolidamento dei duplicati, dei valori mancanti, degli errori e degli *outliers*, ossia di

valori anomali, ma che non sono necessariamente degli errori. Attualmente, solamente i sistemi ETL dispongono di un insieme relativamente completo di funzioni a tale scopo.

Le prime due classi di problemi, (i) e (ii), vengono considerate all'interno del processo di *data cleaning*, mentre la terza (iii) rappresenta l'insieme di operazioni, a cui comunemente si riferisce il termine *preprocessing per data mining*.

Tra i problemi di pulizia dei dati, merita una considerazione particolare l'elaborazione delle stringhe. Questa infatti è una tecnica a basso livello, ma spesso l'unica applicabile, per rendere omogeneo il formato di dati non strutturati. Si noti infatti che, in fase di integrazione di sorgenti multiple, può essere necessario trasformare dati strutturati in dati non strutturati, su cui effettuare il consolidamento dei formati. Si pensi semplicemente, ad esempio, ai diversi formati con cui un indirizzo può comparire all'interno di una base di dati. Esso può avere una struttura, definita dal sistema oppure dall'utente, oppure può essere un dato non strutturato costituito da uno o più campi. Se il tipo dei campi non impone particolari restrizioni, come ad esempio il tipo *stringa*, nessuna assunzione può essere fatta sul contenuto, in particolar modo sul formato. Diventano allora indispensabili delle funzioni di ispezione e di elaborazione di dati a basso livello, in particolare di stringhe. Attualmente, l'unico modo per affrontare questo problema, è rappresentato da programmi scritti ad hoc, molto spesso in C o in Perl. E dal momento che la questione si pone in qualunque progetto che coinvolga sorgenti multiple di dati, nessun sistema visto fino ad ora può considerarsi veramente completo rispetto alla pulizia dei dati. Nello sviluppo del formalismo DPL si è preso atto di questa necessità, mettendo a disposizione alcune primitive di riscrittura delle stringhe. Queste primitive si basano sulla definizione di schemi di costruzione (*pattern*) di generiche stringhe, e sul meccanismo di *pattern matching*. L'argomento sarà approfondito adeguatamente nei paragrafi 2.4 e 3.3.3.

Concludendo, si ricorda che sono innumerevoli gli strumenti che offrono funzionalità utili in fase di preparazione dei dati per l'estrazione di conoscenza. Tra quelli non annoverati c'è *Potter's Wheel* [RH01], un prototipo ETL sviluppato all'Università di Berkeley, il cui vanto principale è il supporto offerto per la rilevazione delle anomalie, oltre all'interattività, entrambe caratteristiche che semplificano il compito dell'addetto alla pulizia dei dati. Un altro progetto ambizioso è rappresentato dal sistema *Mining Mart* [MMart], che unifica tecnologie di data mining con quelle di data warehousing e OLAP (On-Line Analytical Processing). Infine, si è riscontrato

qualche sforzo, nei confronti della preparazione dei dati, anche in alcune estensioni SQL per l'estrazione di modelli di conoscenza. NonStop SQL/MX [CDH99], ad esempio, mette a disposizione delle operazioni di campionamento, mentre MSQL [IV99] possiede primitive di discretizzazione. Nonostante si tratti di operazioni che si limitano alla fase finale di preprocessing, è stato interessante considerare i diversi approcci adottati in sistemi, i quali si avvicinano, per la loro natura dichiarativa, alle nostre proposte di preprocessing tramite DPL.

Capitolo 2

DPL: un formalismo per la preparazione dei dati

Molti sforzi nel campo della ricerca sono rivolti allo studio dei dati, in particolare alle tecniche e agli strumenti per la loro analisi. Ciò è dovuto soprattutto alla crescita del valore strategico dell'informazione, alla crescente concorrenza e all'accumulo di grandi volumi di dati. Tale impegno ha già condotto a molteplici strumenti in grado di ottenere, rappresentare ed utilizzare la conoscenza ricavata dalle collezioni di dati. In particolare, la disciplina del *data mining*, intesa come l'attività centrale del *processo di estrazione di conoscenza (KDD)*, consente di esplorare i dati per individuare i cosiddetti *modelli di conoscenza*, quali le regole di associazione, gli alberi di classificazione, i clusters. Tali modelli di conoscenza costituiscono una formalizzazione, ad un livello di astrazione superiore, di informazioni implicitamente contenute all'interno di una grande quantità di dati. Ha perciò senso considerare l'estrazione di conoscenza come un processo di induzione, o di generalizzazione. Riassumendo, il data mining consente di interpretare e comprendere profondamente i dati, esplorando gli stessi per individuare informazioni significative e potenzialmente utili, e renderle disponibili ai processi decisionali.

Tuttavia, è inutile rivolgere eccessivi sforzi nella ricerca di numerosi modelli, con costi alti in termini di tempo, quando si può estrarre dai dati ben preparati una ben migliore informazione, e in tempi più ristretti, in certi casi attraverso modelli anche più semplici. In questo contesto, una evidente lacuna è rappresentata dallo studio di un'adeguata preparazione dei dati. Sono state sviluppate numerose tecniche a tale proposito, ma sono per lo più tecniche "ad hoc" per scopi molto specifici, note ed utilizzabili solo da pochi esperti. Da un lato, nell'ambito di data mining, si stanno cercando e proponendo soluzioni per ambienti di sviluppo, in modo da estendere ad un pubblico più vasto, anche non esperto, la possibilità di estrarre conoscenza dalle proprie basi di dati. Ma d'altra parte, la preparazione dei dati non è generalmente supportata in

modo adeguato, e richiede ancora che si ricorra a strumenti esterni. Questo rappresenta quindi una limitazione per la diffusione di tali ambienti di estrazione di conoscenza ad un'utenza non esperta. Infatti, il processo è iterativo, e la necessità di preparazione dei dati può insorgere in qualunque momento all'interno di un progetto, richiedendo all'utente di possedere le conoscenze per affrontare il problema ad un basso livello, o quanto meno, di conoscere e avere dimestichezza con altri strumenti. Per questo motivo, tra gli obiettivi fondamentali all'interno del progetto KDDML, c'è l'integrazione di uno strumento per la preparazione dei dati, che sia il più possibile completo.

In questo capitolo verranno introdotti, in maniera informale, gli operatori che costituiscono il linguaggio algebrico DPL. Essi sono il prodotto di una ricerca sulle problematiche della preparazione dei dati, così come dell'analisi dello stato dell'arte, il quale è riassunto nel capitolo precedente. Verrà trattata con particolare enfasi la *pulizia dei dati*, che, nonostante la sua importanza, costituisce il problema più trascurato negli ambienti per applicazioni di KDD. È infatti spesso necessario ricorrere a strumenti esterni appositi, ossia agli *strumenti ETL (Extraction Transformation Loading tools)*. In certi casi non sono sufficienti neanche questi, e l'istanza di problema deve essere affrontato a basso livello tramite dei programmi ad hoc, ad esempio in C. Nel presente lavoro verrà illustrato, anche attraverso degli esempi di utilizzo, come alcuni semplici operatori di *risrittura*, basati sul meccanismo di *pattern matching*, costituiscano un potente strumento di pulizia dei dati.

2.1 La qualità dei dati

La qualità dei dati è una questione cruciale in molte applicazioni, in modo particolare nell'estrazione di conoscenza dai dati. Essa comprende diverse problematiche, tra cui gli errori di immissione dei dati, gli *outliers*¹, i valori mancanti, così come le inconsistenze, le ridondanze, i duplicati, le conversioni dei tipi. Possono

¹ Dati, il cui valore cade al di fuori dei limiti che racchiudono la maggior parte degli altri corrispondenti valori del campione. Non è necessariamente un errore, ed è consigliabile un esame accurato: può portare importanti informazioni.

considerarsi operazioni di qualità anche l'eventuale formattazione o normalizzazione dei dati, di modo che questi ultimi siano strutturati e rappresentati in modo conforme ai requisiti di una particolare applicazione.

Le cause principali di anomalie nei dati possono essere riassunte come:

1. l'assenza di una chiave universale per le diverse basi di dati:

I dati provenienti da sorgenti diverse possono essere stati creati in momenti diversi, da persone differenti, ed usando convenzioni differenti. In questo contesto, decidere quali dati riferiscono lo stesso oggetto reale è importante, ma non è banale. Una compagnia, ad esempio, può possedere informazioni sui suoi clienti in diverse tabelle, perché ogni cliente acquista diversi servizi, i quali sono gestiti da reparti diversi. Se si decide di costruire un unico deposito dati per i clienti della compagnia, lo stesso acquirente può essere riferito in tabelle diverse con nomi differenti, anche se corretti: ad esempio "John Smith", "Smith John" oppure "J. Smith". Questo tipo di discrepanza è nota anche come il *problema d'identità degli oggetti*.

2. l'utilizzo di formati differenti:

Dal momento che, in generale, non sono imposte notazioni standard, i campi dei dati possono anche contenere dati di natura diversa. Ad esempio, il campo *Via* di un indirizzo, può erroneamente contenere il codice fiscale e la città. Oppure possono comparire abbreviazioni o sinonimi per riferire oggetti che sono rappresentati con il nome esteso all'interno di altri records.

3. l'esistenza di errori di immissione dei dati:

A causa di errori di battitura, lo stesso oggetto può essere riferito da più di un record, e qualcuno di questi potrebbe non essere corretto ("John Smith" e "Joh Smith"). Si noti che gli algoritmi

usati per rilevare i duplicati in presenza di errori di immissione sono diversi rispetto a quelli usati per riconoscere che dati in formati diversi sono alternative corrette.

4. la presenza di inconsistenze nei dati:

Una volta che si è riconosciuto che record differenti riferiscono lo stesso oggetto, può succedere che questi contengano informazioni contraddittorie, come ad esempio la data di nascita che non coincide per la stessa persona.

Tutte queste problematiche sono affrontate con la cosiddetta *pulizia dei dati* (in letteratura: *data cleaning/cleansing*), definita come *il processo di miglioramento della qualità dei dati, attraverso la modifica nella forma e nel contenuto*. Durante questa fase, ad esempio, si rimuovono o si correggono i valori dei dati non corretti. Esistono degli strumenti appositi per specificare ed eseguire un tale processo, e sono riferiti in letteratura come *ETL tools* (Extraction, Transformation and Loading). Essi forniscono delle piattaforme potenti per implementare catene di trasformazione di dati, in cui è possibile prelevare flussi di dati da sorgenti arbitrarie, e progressivamente combinare tali flussi attraverso una varietà di operazioni di trasformazione, fino ad ottenere flussi di dati puliti ed opportunamente formattati. I dati risultanti possono successivamente essere caricati nelle basi di dati.

2.1.1 La pulizia dei dati

La fase di *data cleaning*, ossia di pulizia dei dati, che in letteratura viene anche chiamata *data cleansing* oppure *scrubbing*, è la parte più rilevante nel cosiddetto processo ETL (Extraction Transformation Loading) ed ha come obiettivo il miglioramento della qualità dei dati. In essa si affronta il problema di rilevare innanzitutto gli errori e le inconsistenze, per poi procedere alla rimozione di tali errori. Si noti che la qualità dei dati è un fattore di importanza vitale all'interno del processo di estrazione di conoscenza. Possono essere vani tutti gli sforzi, se i dati su cui si lavora dovessero essere anche solo in parte "sporchi". Ciò giustifica la frequenza con cui ci si

imbatte nel motto “Garbage in, garbage out” in letteratura, e l’insistenza sulla questione della qualità dei dati.

La qualità dei dati è un problema che riguarda da un lato singole sorgenti di dati, come ad esempio files o basi di dati, ma assume una notevole importanza quando le sorgenti dei dati sono più di una, sono eterogenee ed è richiesta l’integrazione di tali sorgenti. Nel primo caso gli errori da rilevare e da correggere sono ad esempio quelli dovuti ad immissioni non corrette dei dati, o per errore umano o attraverso immissioni automatizzate dei dati. Anche la mancanza di certi dati oppure valori invalidi sono errori che vanno gestiti. Nel secondo caso, quando è richiesta l’integrazione di più sorgenti, sarà necessario effettuare il data cleaning su ogni singola sorgente, ma ciò non basta. Le sorgenti conterranno tipicamente dati ridondanti, e molto probabilmente rappresentati in maniera non uniforme. Si tratterà quindi di eliminare le informazioni duplicate e di consolidare le rappresentazioni diverse di dati.

Il processo ETL nominato sopra, come dice il nome, comprende l’estrazione di dati da più sorgenti, la trasformazione di tali dati sia nello schema che nel contenuto, in modo da consolidare le informazioni ridondanti e rappresentate in maniera non uniforme. Infine i dati delle diverse sorgenti sono integrati, filtrati ed aggregati, e memorizzati in un unico deposito dati. Questo è un processo tipicamente legato all’ambito *data warehouse*, ma risulta necessario ogni qual volta si vogliano integrare multiple sorgenti di dati. Per questo motivo useremo spesso questo termine (ETL), senza presumere che il processo di estrazione di conoscenza abbia necessariamente come punto di partenza un deposito dati, già consolidato, di un data warehouse.

Attualmente è disponibile una grande quantità di strumenti ETL, anche molto diversi fra loro, ed anche ogni data warehouse offre almeno qualche funzionalità ETL. Ciononostante, chiunque debba affrontare un problema di preparazione dei dati all’interno di un progetto in un ambito specifico, difficilmente troverà uno strumento in grado di coprire tutte le sue necessità. Il supporto che offrono, ad esempio, i sistemi data warehouse ed i sistemi di basi di dati federate è indirizzato prevalentemente alle funzionalità di *integrazione* e di *trasformazione di schema*, mentre gli strumenti di *trasformazione dei dati* è piuttosto limitato. D’altra parte, gli strumenti ETL, che focalizzano esclusivamente sul processo di *data cleaning*, offrono funzionalità generiche, e difficilmente riescono a supportare in modo completo la preparazione dei dati per scopi specifici, come può essere il data mining.

La comunità di ricerca si è occupata intensamente di traduzione e di integrazione di schema, ma non si riscontra un grande interesse nei confronti della pulizia dei dati, nonostante l'elevato numero di strumenti commerciali testimoni l'importanza e la difficoltà di questo problema.

Diversi autori si sono occupati del problema dell'identificazione ed eliminazione di duplicati (*merge-purge problem* in letteratura) [GFSS00] [GFS00] [HS98] [M00]. Qualche gruppo di ricerca si è concentrato su problemi generali non strettamente limitati al data cleaning, ma che hanno una certa rilevanza in questo campo, come ad esempio approcci particolari e specializzati nel data mining [SA95], oppure trasformazioni di dati basati su *schema matching* [ACM99]. Recentemente è stato proposto e analizzato un trattamento più uniforme del data cleaning, che coprisse diverse fasi di trasformazione, operatori specifici e la loro implementazione [GFSS00] [RH01].

2.1.2 Problemi di dati “sporchi”

Vengono di seguito elencati i problemi principali di anomalie nei dati, e che richiedono necessariamente una trattazione. Si cerca quindi di individuare le funzionalità che potrebbero fornire un supporto nella risoluzione di tali problemi, e dopo avere introdotto i costrutti del linguaggio, saranno proposte delle soluzioni secondo l'approccio DPL.

1.) Valore multiplo in un singolo attributo

Es: Name = “Curt Davis DBA CD Design”
Address = “1201 S.Hathaway Street Apartment 12A”

Questo esempio rende subito evidente la necessità di uno strumento in grado di elaborare le stringhe, come dato di basso livello. Si noti che non basterà semplicemente individuare le sottostringhe separate dal carattere *spazio*, ma sarebbe opportuno disporre di un meccanismo di riconoscimento degli schemi di

costruzione delle stringhe. Inoltre sono necessari dei costrutti per modificare lo schema di una tabella relazionale, modificando, aggiungendo e rimuovendo gli attributi.

2.) Valore spezzato: attributi senza formato e di lunghezza fissa (tipicamente di tipo stringa) possono provocare lo spezzamento del valore immesso in due campi di inserimento che compaiono adiacenti.

Es: Address1 = “1201 S.Hathaway Street Apart”
Address2 = “ment 12A”

Anche in questo caso sarebbe utile un meccanismo di confronto tra stringhe, che non si limiti ad una elementare operazione di uguaglianza. Si potrebbe, ad esempio, verificare che il valore dell’indirizzo sia “simile” ad una stringa formata da tre componenti: la prima costituita da un numero di quattro o cinque cifre, la seconda formata da una sequenza di parole che indica la via, e la terza che corrisponde al numero civico, magari preceduto da qualche parola che specifica tale numero.

3.) Valore non corrispondente ad un certo attributo

Es: State = “England”

4.) Errore di digitazione

Es: City = “New Yrok”

Gli esempi 3. e 4. mostrano che è necessario disporre di primitive per verificare che il valore di un certo attributo rientri in un insieme predefinito di possibili valori. Anche questo problema richiede un controllo sul dato di tipo stringa, ma qui è sufficiente un elementare confronto di uguaglianza. Per individuare errori di digitazione potrebbe anche essere di una certa utilità un ordinamento, basato sulla frequenza con cui ricorrono i diversi valori.

5.) Valori fuori dal dominio

Es: Salary = “-10”
Age = “-27”

È necessario effettuare dei controlli sul dato *valutato*. Per questa necessità sono sufficienti delle tradizionali interrogazioni in stile SQL. Però, piuttosto che ottenere come risultato i dati che soddisfano (o meno) una certa condizione, sarebbe di maggiore utilità poter “marcare” in qualche modo tali dati.

6.) Valori mancanti o sconosciuti

Es: Occupation = “”
Salary = “Unknown”

Per individuare questi tipi di anomalie possono servire, sia il controllo sullo schema di costruzione di una stringa, che l’uguaglianza semplice. Come trattare, ed eventualmente sostituire, questi valori è un’altra questione. A tale scopo

possono comunque fornire un supporto le funzionalità di analisi e quelle di data mining, già presenti nel sistema.

7.) Inconsistenze, vincoli d'integrità

Es: *Codice fiscale* duplicato
Età e DataNascita inconsistenti

Sarebbe utile poter “marcare” le tuple che contengono gli stessi valori rispetto ad un sottoinsieme degli attributi. In questo modo si possono ispezionare tali presunti duplicati, e verificare l'effettiva ridondanza, prima di procedere al loro consolidamento.

Le inconsistenze tra gli attributi all'interno delle singole tuple possono essere rilevate con la valutazione di espressioni multi-attributo, disponibili in SQL, abbinate al meccanismo di marcatura dei dati anomali.

8.) Formato e/o

9.) unità di rappresentazione di un attributo variano nel tempo e nelle diverse sorgenti

Es: Sex: 1,0 vs M,F
Misura: 1,2,3 vs A-B

Le discrepanze si possono individuare con i controlli di uguaglianza, ma sarebbe pratico convertire allo stesso tempo i formati, ottenendo dei valori uniformi. Allora si rivelerebbe utile una primitiva che sostituisca ogni valore non uniforme, ma significativo, con un altro valore corrispondente. Bisogna tenere in

considerazione anche la necessità di ottenere il nuovo dato come risultato di un'espressione che coinvolga anche altri attributi della tupla.

10.) Modifica della semantica di un attributo nel corso del tempo

Es: Bonus = "400" : 400 dollari, prima del 1986

Bonus = "10" : 10% dello stipendio dopo il 1986

Questo problema, per essere trattato adeguatamente, richiede necessariamente l'informazione *anno*. A partire da tale ipotesi, si può costruire un nuovo attributo *NewBonus*, il cui valore è quello di *Bonus* moltiplicato per lo stipendio, nel caso in cui il dato risale ad un anno successivo al 1986, e quello di *Bonus* altrimenti. Se l'anno non è disponibile, né tra i dati, né tra la documentazione, solo l'ispezione dei dati e l'analisi può fornire qualche aiuto. Ma, a meno che i dati con semantiche diverse non abbiano valori discostati in modo evidente, non si può fare affidamento su di essi, e i dati sono praticamente inutilizzabili per il data mining.

11.) Duplicati

Una nota particolare meritano i cosiddetti *duplicati*, ossia i dati che si riferiscono allo stesso oggetto reale, ma che non sono necessariamente uguali, né simili. Nel caso in cui esiste un codice unico, oppure un insieme di attributi che identificano univocamente gli oggetti del dominio, è facile individuare i duplicati attraverso un controllo di uguaglianza. Il consolidamento è una questione a parte, in cui vanno considerate le discrepanze tra diverse rappresentazioni, ed eventualmente le inconsistenze.

Duplicati possono non essere identici perché:

- le cose cambiano nel tempo (ad esempio l'indirizzo)
- i valori non sono corretti
- i dati provengono da sorgenti diverse (con attributi diversi, oppure attributi con rappresentazioni diverse o semantiche diverse)
- i dati possono essere rappresentati in maniera differente (sinonimi, acronimi, abbreviazioni, ordine delle componenti, ...)
- ...

Es:	“Dr”	vs	“Drive”
	“KFC”	vs	“Chicken, Kentucky Fried”
	“C. E. Davis“	vs	“Curtis Edward Davis“

L'identificazione e l'eliminazione dei duplicati, note in letteratura con il termine *merge-purge problem*, rimane tuttora un problema aperto, nonostante diversi autori se ne siano occupati [GFSS00] [GFS00] [HS98] [M00].

Si noti, in ogni caso, che per agevolare l'individuazione di questo problema, esso va trattato solo dopo la pulizia di tutte le anomalie elencate precedentemente.

2.2 Requisiti del sistema

Lo studio di ricerca sulla preparazione dei dati per l'estrazione di conoscenza dai dati, assieme ad un'analisi degli strumenti esistenti, hanno permesso di individuare un insieme di requisiti per lo strumento proposto in questo lavoro di tesi, tenendo in considerazione anche il contesto in cui esso si inserisce. In particolare, l'approccio al data cleaning, richiede alcune considerazioni particolari. Esso dovrebbe essere supportato da strumenti in modo da limitare l'ispezione manuale e lo sforzo di

programmazione dell'addetto, così come dovrebbe essere estensibile ad eventuali sorgenti addizionali. Inoltre, il data cleaning non va eseguito in modo isolato, ma assieme a trasformazioni di dati e di schema. Le modifiche non devono in generale mai sostituire i valori originali dei dati, in quanto in un secondo momento potrebbe essere rivalutata un'informazione che è stata eliminata o alterata. Ad esempio, potrebbero possedere un certo contenuto informativo gli errori e gli *outliers*, eliminati in fase di preparazione dei dati, in quanto non sono trattabili da un alcuni algoritmi di data mining; essi infatti, nelle reti neurali, possono rendere inutilizzabili altri dati.

Ad un alto livello sono stati individuati i seguenti requisiti:

1) Regole di riscrittura

Come si è visto nel paragrafo precedente, dopo l'importazione di dati esterni nel sistema, e soprattutto in seguito all'integrazione di sorgenti multiple, nessuna assunzione può essere fatta sul contenuto, in particolar modo sul formato dei dati. Diventano allora indispensabili delle funzioni di ispezione e di elaborazione dei dati a basso livello, ovvero della sequenza di caratteri che li rappresenta. Dal momento che la questione si pone in qualunque progetto, si è preso atto di questa necessità, mettendo a disposizione alcune primitive di riscrittura delle stringhe. Prendendo spunto proprio dagli operatori del Perl, a cui si ricorre frequentemente in questi casi, tali regole di riscrittura si basano sulla definizione di schemi di costruzione (*pattern*) di generiche stringhe, e sul meccanismo di *pattern matching*, che saranno trattati in modo più approfondito nel paragrafo 2.4 .

2) Costrutti tipo SQL

Gli operatori dell'algebra relazionale sono necessari in fase di integrazione, per effettuare trasformazioni a livello dello schema: per rimuovere o rinominare attributi, per definirne di nuovi, a partire dai valori degli altri attributi della relazione. Gli operatori di restrizione e di proiezione invece servono per selezionare e ridurre la dimensione dei dati, che costituiscono l'ingresso degli algoritmi di data

mining. Infine, le funzioni di aggregazione ed i costrutti in stile SQL, associati ad un meccanismo di marcatura, possono anche fornire uno strumento di analisi e di ispezione dei dati, fondamentali sia prima che durante la preparazione dei dati al data mining.

3) Funzioni di preprocessing ad hoc per il data mining

Nel sistema sono indispensabili tutte quelle funzioni di trasformazione che sono applicate nell'ultimo stadio della preparazione. A queste appartengono, ad esempio, la discretizzazione, la normalizzazione, la binarizzazione, ma anche il campionamento. Si tratta delle trasformazioni effettuate sui dati già integrati e ripuliti dalle anomalie, e che adeguano gli stessi alle specifiche degli algoritmi di data mining. Queste funzionalità sono messe a disposizione da ogni ambiente di sviluppo per applicazioni di data mining, ma non sono necessariamente fornite dagli strumenti ETL.

4) Strumenti di visualizzazione e di statistica

Nella fase preliminare di analisi dei dati è necessario disporre di strumenti in grado di visualizzare, attraverso delle opportune metafore grafiche, tutte le caratteristiche rilevanti dei dati. Ad esempio, è utile conoscere la distribuzione dei dati, ricercare delle possibili relazioni tra gli oggetti, oppure le relazioni tra gli attributi di un singolo oggetto, attraverso l'ausilio di grafici, quali gli istogrammi, gli ortogrammi, gli aerogrammi, i diagrammi cartesiani, ecc. Inoltre, sarebbe interessante visualizzare graficamente anche le diverse anomalie rilevate nei dati, sfruttando il meccanismo di marcatura.

5) Modelli di conoscenza

I modelli di conoscenza rappresentano un potente strumento di supporto per la preparazione dei dati, in particolare nella pulizia dei dati. Si noti, infatti, come ad esempio la clusterizzazione possa rilevare, sotto certe condizioni, alcune anomalie nei dati, come gli *outliers*, o i valori mancanti. Le regole di associazione potrebbero aiutare nella scoperta di duplicati, e gli alberi di decisione, costruiti su dei *test set* affidabili, stimerebbero i valori in corrispondenza di dati mancanti. Per ora sono state fatte solamente interessanti ipotesi, ma non sono ancora affermate le strategie di *data cleaning* che adottano i modelli di conoscenza. In ogni caso, il sistema dispone già di questa promettente funzionalità.

6) Meccanismo per marcare/evidenziare valori “non conformi”

Un meccanismo di marcatura si rivela di fondamentale importanza, per risolvere diversi problemi. Innanzitutto, esso serve per ispezionare i dati: combinato con il meccanismo del *pattern matching*, è possibile verificare che tutti i dati di tipo stringa rispettino un certo schema di costruzione, evidenziando gli eventuali dati “non conformi” ad un *pattern* specifico, così come serve anche ad evidenziare i valori mancanti e sconosciuti. Successivamente i dati marcati possono essere rintracciati velocemente, e trattati in modo opportuno. Sempre in fase di analisi, serve anche la marcatura basata su una condizione di *valutazione di espressioni*, possibilmente multi-attributo, per rintracciare ad esempio le violazioni dei vincoli di integrità.

Inoltre, attraverso la marcatura, si può anche tenere traccia delle eccezioni che si verificano durante la valutazione delle operazioni DPL. Nei casi in cui i dati in ingresso non sono “conformi” alle specifiche del costrutto, e non è definita la semantica dell’operatore, allora la marcatura consente di estendere tali dati con tutte le informazioni utili per riconoscere e gestire successivamente l’eccezione.

2.3 Introduzione agli operatori DPL

In questo paragrafo vengono enumerati i costrutti che rappresentano le operazioni primitive del linguaggio DPL. Se ne darà quindi una descrizione breve ed informale, per introdurre in modo graduale la soluzione proposta in questo lavoro di tesi. La semantica verrà trattata più in dettaglio nel capitolo 3. Nel paragrafo 2.5 saranno inoltre riportati alcuni esempi, allo scopo di illustrare l'utilizzo di tali operatori.

Nell'introduzione alla sintassi dei costrutti, si userà il carattere *Corsivo* per indicare i parametri attuali, quali ad esempio la tabella relazionale in ingresso, l'attributo su cui l'operatore agisce, ma anche delle costanti o variabili che contengono le specifiche per eseguire l'operazione associata al costrutto. Lo stile *arial* è usato per abbreviare una componente sintattica del costrutto, che tramite una definizione formale risulterebbe troppo estesa, e poco intuitiva. Il carattere MAIUSCOLO infine riporta le componenti sintattiche in modo preciso.

2.3.1 Il costrutto di riscrittura

```
APPLY RWRULES IN Tabella TO Attributo
BEGIN
    sequenza di regole di riscrittura
END
```

Con questo costrutto è possibile specificare, contemporaneamente, una sequenza di regole di riscrittura, che saranno eseguite nello stesso ordine in cui esse compaiono. Tutte le regole che compongono la sequenza si riferiscono allo stesso attributo, che viene trattato, ad un basso livello, ossia come una sequenza di caratteri.

Sono previsti tre tipi di regole di riscrittura. La *regola di sostituzione*, riferita in seguito con il termine *ruleS*, permette di sostituire il valore dell'attributo specificato, con una nuova stringa, costruita a partire da quella originale. A tale scopo si sfrutta il meccanismo di pattern matching, che verrà descritto meglio nel paragrafo 2.4. La *regola di marcatura*, ovvero *ruleM*, consente di marcare l'attributo specificato nel costrutto, in

tutte le tuple della tabella in cui il pattern matching non ha avuto successo. Infine, tramite la *regola di traduzione* (*ruleT*), si possono “tradurre” tutte le occorrenze di una sottostringa all’interno dell’attributo. Tale sottostringa è specificata attraverso un’espressione regolare (vedi par 2.4), che ne descrive lo schema di costruzione, e la sottostringa di sostituzione corrispondente può essere formata da un valore costante, oppure anche da una sequenza di caratteri e di *variabili di raggruppamento* (vedi par 2.4).

Il costrutto di riscrittura si rivela di notevole utilità durante tutta la fase di pulizia dei dati, sia per individuare dei valori “non conformi”, in questo caso le stringhe che non rispettano un dato *pattern*, sia per uniformare i formati.

Si noti comunque che c’è una limitazione importante, in quanto la pulizia e la marcatura possono trattare solamente le anomalie all’interno di un singolo attributo.

2.3.2 La marcatura

MARK *Attributo* IN *Tabella*

IF (*condizione di marcatura*)

[WITH CODE *Codice_di_marcatura*]

Questo secondo costrutto di marcatura si distingue dalla regola per la marcatura *ruleM*, in quanto permette di specificare la condizione attraverso la valutazione di un’espressione, piuttosto che attraverso il *pattern matching*. L’attributo non viene trattato a basso livello, ossia come sequenza di caratteri, ma come un dato che va valutato all’interno di un’espressione booleana, la quale costituisce la condizione di marcatura. Un’altra differenza importante rispetto alla regola *ruleM*, e che ha reso necessario questo ulteriore costrutto per la marcatura, è rappresentata dalla possibilità di specificare una condizione multi-attributo. Infatti, per controllare ad esempio che siano soddisfatti i vincoli d’integrità, è indispensabile un controllo che, per ogni tupla, coinvolga diversi attributi. È inoltre possibile assegnare un codice alle tuple marcate con questo costrutto.

2.3.3 La divisione

```
DIVIDE Attributo IN Tabella  
IF ( condizione di divisione )  
PUT INTO Nuovo_attributo
```

Il costrutto DIVIDE appartiene alla classe delle *trasformazioni di schema*, in quanto effettua soprattutto una modifica a livello di schema relazionale. Esso è di particolare utilità in fase di consolidamento di dati provenienti da sorgenti diverse.

Innanzitutto, la tabella in ingresso viene estesa con un nuovo attributo, il cui nome segue il tag *PUT INTO*. Se la condizione di divisione è verificata, il valore associato all'attributo di origine viene spostato e associato al nuovo attributo creato, altrimenti il valore rimane associato all'attributo di origine.

La condizione, in base alla quale il contenuto di un attributo viene distribuito tra due colonne, può essere rappresentata da un'espressione booleana formata da operatori su stringhe, operatori matematici ed anche logici, ma non è prevista la condizione multi-attributo. Inoltre, la condizione prevede anche un controllo di *pattern matching*.

2.3.4 Lo split

```
SPLIT Attributo IN Tabella  
AT EspressioneRegolare  
PUT INTO NuovoAttributo1 AND NuovoAttributo2  
[EXCEPTION [Codice_di_marcatatura]]
```

Anche SPLIT è un costrutto che modifica lo schema relazionale della tabella, ed appartiene quindi alla classe delle *trasformazioni di schema*. Vengono introdotti due nuovi attributi, e la sequenza di caratteri, che rappresenta il valore dell'attributo specificato in ingresso, è spezzata e distribuita sui due nuovi attributi. Il punto di "spezzamento" viene identificato come la prima occorrenza di una sottostringa specificata dall'utente attraverso una espressione regolare, e tale sottostringa apparterrà

interamente alla seconda parte del taglio. Se il punto di spezzamento non dovesse essere individuato in qualche tupla, è possibile richiedere la sua marcatura, eventualmente assegnando un codice di identificazione.

2.3.5 La fusione

```
MERGE Attributo1 AND Attributo2  
IN Tabella  
PUT INTO Nuovo_Attributo
```

Come il costrutto precedente, MERGE aggiunge allo schema relazionale della tabella un nuovo attributo. Il valore che esso assume è formato dalla fusione dei due attributi specificati, ovvero dalla concatenazione dei caratteri che costituiscono il loro valore. Questi attributi non vengono rimossi dalla tabella. Anche MERGE è una *trasformazione di schema*.

2.3.6 Rinominazione, rimozione, creazione di un attributo

```
RENAME Attributo IN Tabella  
AS Nuovo_Attributo
```

```
REMOVE Attributo IN Tabella
```

```
NEWATTRIBUTE Nuovo_attributo IN Tabella  
[ IS espressione ]
```

I tre costrutti RENAME, REMOVE e NEWATTRIBUTE sono le tre *trasformazioni di schema* di base, indispensabili in ogni linguaggio per basi di dati. Per questo si è preferito nominarli a parte, nonostante essi siano già disponibili tra gli operatori SQL.

L'espressione per la costruzione del nuovo valore da inserire in ogni tupla può essere formata da operatori matematici, su stringhe, e di congiunzione logica. Oltre ai valori costanti possono comparire variabili che rappresentano gli altri attributi della tabella

2.3.7 Il ripiegamento

FOLD Attributo1 ON Attributo2 IN Tabella

Il costrutto FOLD, a livello di schema relazionale, comporta la riduzione del numero di colonne di una tabella, e contemporaneamente un raddoppio del numero di righe. In particolare, viene rimossa la colonna corrispondente al primo attributo specificato, ma sarebbe improprio dire che viene rimosso l'attributo: infatti, il valore del primo attributo viene inserito in corrispondenza del secondo. Ogni tupla viene in questo modo replicata, ed ogni coppia si distinguerà solamente per il valore associato al secondo dei due attributi specificati nel costrutto FOLD.

2.3.8 Ordinamento per frequenza

SORT Tabella BY FREQUENCY

ON Attributo

Questo costrutto può servire in fase di analisi dei dati, in particolare per ispezionare la distribuzione dei dati, quando non è possibile farlo in modo grafico. Tramite esso vengono ordinate le tuple della tabella in base alla frequenza con cui compaiono i valori dell'attributo specificato. In questo modo è anche possibile scoprire, in mezzo ad una grande quantità di dati, alcuni dati anomali. Questi ultimi potrebbero essere degli *outliers*, ossia valori che si discostano molto dagli altri, ma che non sono necessariamente degli errori. Ciononostante è importante rilevarli, perché possono alterare i risultati degli algoritmi di data mining, tanto da renderli addirittura non significativi. Inoltre si possono individuare alcuni errori di digitazione o valori associati

ad un attributo non corrispondente, sempre che queste anomalie si verifichino in modo eccezionale.

Alla tabella riordinata viene anche aggiunto un nuovo campo, il quale riporta, per ogni tupla, la frequenza con cui è stato riscontrato, all'interno dell'intera tabella, il valore dell'attributo.

2.3.9 La marcatura e la fusione di duplicati

```
MARK DUPLICATES IN Tabella ON ( sequenza di attributi )  
[WITH CODE Codice_di_marcatura]
```

```
MERGE DUPLICATES IN Tabella ON ( sequenza di attributi )  
[WITH CODE Codice_di_marcatura]
```

Questi due costrutti verificano l'eventuale uguaglianza, rispetto ad un insieme di attributi, tra tuple adiacenti. Si noti, che essi richiedono in ingresso una tabella ordinata rispetto a tali attributi, perciò devono essere preceduti da un ordinamento per mezzo di un tradizionale comando SQL.

Il primo costrutto permette di marcare i duplicati individuati in questo modo, mentre il secondo elimina dalla tabella, per ogni tupla, tutti i duplicati successivi. Il codice di marcatura può essere utilizzato nel costrutto MERGE come condizione aggiuntiva per considerare un duplicato come una tupla da rimuovere. Nel costrutto MARK invece consente di assegnare un'identificazione alla marcatura.

2.3.10 Il campionamento

```
SAMPLE Tabella metodo di campionamento  
USE algoritmo
```

Il campionamento è una trasformazione di riduzione verticale, che nonostante la semplicità della sua esecuzione, è delicata come questione. Essa è indispensabile per poter effettuare l'estrazione di modelli di conoscenza da collezioni di dati, che avrebbero altrimenti dimensioni intrattabili. Però va considerato che il campionamento può anche alterare i risultati della fase di data mining, se esso non è preceduto da un'analisi della distribuzione dei dati, delle correlazioni tra tuple e tra attributi. È inoltre necessario tenere presente quale è lo scopo di questa riduzione verticale, ossia quale è l'uso che si intende fare dei dati selezionati. Sono previste alcune varianti del costrutto, a seconda del tipo di campionamento e della strategia di elaborazione.

2.3.11 La normalizzazione

NORMALIZE_metodo (sequenza di attributi)

IN Tabella [EXCEPTION [Codice_di_marcatura]]

Questo costrutto consente di ridimensionare i valori di uno o più attributi, e rappresenta una forma di riduzione della dimensione dei dati. Come il campionamento è una trasformazione applicata ai dati già integrati e puliti, e serve per adeguare la rappresentazione degli stessi alle specifiche degli algoritmi di data mining. Sono previsti diversi metodi di normalizzazione, e in alcune di queste varianti, in cui si possono verificare delle eccezioni durante l'elaborazione, è possibile marcare le tuple che le hanno sollevate.

2.3.13 La discretizzazione

DISCRETIZE Attributo IN Tabella

tipo, algoritmo e parametri di discretizzazione

Anche la discretizzazione effettua una riduzione della dimensione dei dati, ed appartiene alle trasformazioni di preprocessing apposite per il data mining. Essa consiste nel ridurre il numero di possibili valori che un attributo può assumere, e si può anche vedere come una forma di generalizzazione di un concetto. Bisogna allora tenere presente che questa trasformazione potrebbe comportare la perdita di informazione di un certo rilievo per l'accuratezza dei risultati di data mining.

Esistono molteplici varianti, a seconda del tipo dell'attributo, della distribuzione e delle correlazioni presenti tra i dati.

2.3.14 I costrutti SQL

Gli operatori di interrogazione SQL si rivelano utili innanzitutto per effettuare delle riduzioni di dimensione, attraverso la proiezione, la restrizione e l'aggregazione, ma anche per ispezionare i dati ed effettuare analisi statistiche elementari. In fase di consolidamento è inoltre necessario disporre della giunzione, mentre l'operatore di ordinamento può essere di ausilio, sia per ispezionare i dati, che per applicare i costrutti di marcatura e di fusione dei duplicati. Sarebbe però auspicabile estendere la condizione *WHERE* dell'operatore SQL *SELECT* con una *condizione di marcatura/eccezione*.

2.4 Introduzione alle espressioni regolari

Una delle caratteristiche del linguaggio di preparazione dei dati DPL è sicuramente quella di poter operare in modo estremamente flessibile sulle sequenze di caratteri. Per sfruttare appieno la potenza espressiva delle regole di riscrittura, è però necessario imparare a conoscere le *espressioni regolari* e a capire come funziona il meccanismo del *pattern matching*.

Ciò che ci permette di fare la maggior parte dei linguaggi di programmazione è di verificare che una certa sottostringa prefissata sia contenuta in una stringa più lunga. Il meccanismo di *pattern matching* invece ci consente di generalizzare questa operazione,

ricercando all'interno di una stringa, non una sottostringa ben precisa, ma una generica sottostringa, la cui struttura viene descritta utilizzando una sintassi particolare. A questo scopo, le *espressioni regolari* permettono di descrivere uno schema di costruzione di una stringa, ovvero un *pattern*. L'operazione di *pattern matching*, a questo punto, consente di verificare se una stringa rispetta un dato schema di costruzione, ovvero se appartiene ad una classe di stringhe, descritta mediante una espressione regolare.

Per chiarire meglio questo meccanismo, vediamo subito un esempio elementare. Supponiamo di voler “descrivere” tutte le stringhe composte secondo il seguente schema: una sequenza arbitrariamente lunga di cifre, seguita da un carattere di spaziatura o tabulazione, che termina con una parola composta da caratteri qualsiasi. Con una espressione regolare possiamo descrivere questa composizione di stringa nel seguente modo:

```
\d+\s.+
```

I primi due caratteri “\d” indicano la presenza di un carattere numerico (0, 1, 2, ..., 9); il carattere “+” indica che il carattere numerico può presentarsi una o più volte. La sequenza “\s” indica un qualsiasi carattere di spaziatura o di tabulazione. Infine il punto “.” indica un carattere qualsiasi, e il simbolo “+”, come prima, sta ad indicare che questo tipo di carattere può ricorrere una o più volte.

Questa espressione regolare descriverà quindi stringhe quali: “1234 pippo”, “1 2”, “1 ab\$%\&xy”. Le seguenti stringhe invece non risultano descritte dalla precedente espressione regolare: “a b”, “pippo”, “albero casa”, “1+3=4”.

L'operazione di *pattern matching* consiste quindi nel verificare che una stringa o una sua sottostringa iniziale sia costruita secondo uno schema descritto da una espressione regolare. Si noti che, come effetto secondario, è possibile memorizzare, attraverso il raggruppamento, le sottostringhe che rispettano un dato schema di costruzione. Questo permette di utilizzarle nella costruzione di una nuova stringa, oppure di verificare delle condizioni su sottostringhe, le quali non sono specificate con esatta precisione, ma in modo molto flessibile attraverso uno schema di costruzione.

La seguente tabella descrive sinteticamente i termini che possono comporre una espressione regolare all'interno del formalismo DPL:

.	qualsiasi carattere escluso il new line (" $\backslash n$ ")
[X]	corrisponde uno dei carattere nell'insieme X
[^ X]	non corrisponde uno dei carattere nell'insieme X
$\backslash d$	una cifra qualsiasi; equivalente a "[0-9]"
$\backslash D$	un carattere che non sia una cifra; equivalente a "[^0-9]"
$\backslash w$	un carattere alfanumerico; equivalente a "[a-zA-Z0-9]"
$\backslash W$	un carattere non alfanumerico; equivalente a "[^a-zA-Z0-9]"
$\backslash s$	un carattere di spaziatura (spazio, tabulazione, new line, ecc.)
$\backslash S$	un carattere non di spaziatura
$\backslash n$	il carattere <i>new line</i>
$\backslash r$	il carattere return (ritorno carrello)
$\backslash t$	il carattere di tabulazione
$\backslash b$	confine di una parola
$\backslash \emptyset$	null, il carattere nullo
$\backslash xHH$	Carattere con codice esadecimale HH
\backslash	il prossimo carattere è considerato come tale, non come un metacarattere
(s)	raggruppamento della sottostringa identificata dall'espressione regolare s
-	intervallo all'interno di insieme di caratteri, ad esempio "[0-9]"
x^*	il carattere identificato da x ripetuto 0 o più volte
x^+	il carattere identificato da x ripetuto una o più volte
$x\{n\}$	il carattere identificato da x ripetuto n volte
$x\{n,\}$	il carattere identificato da x ripetuto almeno n volte
$x\{n,m\}$	il carattere identificato da x ripetuto da n a m volte
pippo	la stringa "pippo"
	alternativa, ad esempio "aalbb" è la stringa "aa" oppure la stringa "bb"
\$	la stringa termina con l'espressione regolare precedente

Tab. 2.1: Termini per la composizione di una espressione regolare in DPL

Seguono alcuni esempi di chiarificazione:

ciao\.

corrisponde a *ciao* seguito da un punto

.iao

	corrisponde sia a 'ciao' che a 'miao'
test	corrisponde a tutto ciò che comincia per test, ovvero 'testo', 'testimone', ma non 'il test'
me\$	corrisponde esattamente a 'me'
gatto topo	corrisponde sia a 'gatto' che a 'topo'
signor(ile)	corrisponde a 'signori' e a 'signore' (e la variabile di raggruppamento (\$1) contiene <i>i</i> oppure <i>e</i>)
[aeiou]	corrisponde ad una (sola) vocale
[a-zA-Z0-9]	corrisponde ad un (solo) carattere alfanumerico

Alcuni esempi con i quantificatori: * + {n} {n,} {n,m}

ma+	corrisponde a 'ma', 'maaa', 'maaaaaaa'
ma*	corrisponde a 'm', 'maaa'
(yada){2,}	corrisponde a 'yada yada ', 'yada yada yada yada '

Un esempio con i metacaratteri: \d \s \w ...

$(\d\d\d)\s*\d\d\d[-]\{0,1\}\d\d\d\d$

corrisponde a un numero telefonico americano, ovvero 3 cifre racchiuse in parentesi, seguite da zero o più spazi, un prefisso a 3 cifre, seguito ancora da una (o nessuna) sequenza formata da un trattino e uno spazio, per finire con quattro cifre.

Un esempio con i raggruppamenti:

`(([a-z]+)@(cli\.di\.unipi\.it))`

corrisponde ad esempio a `'valentim@cli.di.unipi.it'`, e dopo aver effettuato il pattern matching con questa ultima stringa, le variabili di raggruppamento $\$i$ contengono i valori:

$\$1 = 'valentim@cli.di.unipi.it'$

$\$2 = 'valentim'$

$\$3 = 'cli.di.unipi.it'$

2.5 Esempi

In questo paragrafo si intende fornire degli esempi di qualche tipico problema nell'ambito della preparazione dei dati, assieme alla rispettiva soluzione in termini degli operatori DPL.

Il primo esempio illustrerà l'utilizzo del costrutto di riscrittura, lasciando intuire la sua forte potenzialità. Seguiranno degli esempi che chiariscono meglio la semantica delle trasformazioni di schema, e dei costrutti di rilevazione dei duplicati. Infine si descrive un semplice caso di integrazione di sorgenti multiple.

2.5.1 Applicazione delle regole di riscrittura

Esempio 1.

Supponiamo che sia data una tabella con i seguenti valori dell'attributo *Data*:

Data
25/10/78
2/17/81,
20th March 2003
18.09.2000
in data 2/7/18
08/19/1950
32.05.75
8/07/328
...

e sia richiesto di risolvere il seguente

Problema:

a) Si eliminino caratteri di spaziatura e sequenze di caratteri non significativi adiacenti agli estremi della data;
si sostituiscano caratteri di punteggiatura con il carattere '/';
si aggiunga uno zero al giorno e al mese quando la data non è nel formato 'xx/yy/anno'.

b) Si modifichi la data "xx/yy/aa" in "yy/xx/aa" quando $yy > 12$.

c) Quando la data è nel formato "xx/yy/aa", se $aa > 30$ la si cambi in "xx/yy/19aa"
e se $aa \leq 30$ la si cambi in "xx/yy/20aa".

d) Infine controllare la data rispetti sempre il formato "xx/yy/aaaa" e che $xx \leq 31$.

Soluzione:

```

APPLY RWRULES IN 'tabella' TO 'data'
BEGIN
/** riscrittura a) */
  << \D* (\d{1,2}) [./] (\d{1,2}) [./] (\d{1,4}) \D* >> INTO $1/$2/$3 EXCEPTION;
  << (\d) / ([\d/]*) >> INTO 0$1/$2 ;
  << (\d\d) / (\d) / ([\d/]*) >> INTO $1/0$2/$3 ;
  << (\d\d) / (\d\d) / (\d) >> INTO $1/$2/0$3 ;

/** riscrittura b) e d) */
  /** se l'anno avesse tre cifre
  << (\d{2}) / (\d{2}) / ((\d\d){1,2}) >> INTO IF( $2 > 12 )
  THEN $2/$1/$3 EXCEPTION;

/** riscrittura c) *
  << (\d{2}/\d{2}/) (\d{2}) >> INTO IF( $2 > 30 ) THEN $119$3
  ELSE $120$3 ;

/** riscrittura d) */
  << ([0-2] [0-9] [\d/]*) | (3 [0,1] [\d/]*) >> EXCEPTION ;
END

```

Le regole vengono applicate nell'ordine in cui compaiono, e di seguito sono riportate le trasformazioni che la tabella subisce, man mano, nel corso dell'elaborazione. Il grassetto indica i valori che sono stati marcati, quando è fallito il pattern matching in corrispondenza di una regola contenente il tag *EXCEPTON*.

Tabella di partenza:

Data
25/10/78
2/17/81,
20th March 2003
18.09.2000
in data 2/7/18
08/19/1950
32.05.75
8/07/328
...

a)



Data
25/10/78
2/17/81
20th March 2003
18/09/2000
2/7/18
08/19/1950
32/05/75
8/07/328
...

a),b)



Data
25/10/78
17/02/81
20th March 2003
18/09/2000
02/07/18
19/08/1950
32/05/75
08/07/328
...

c),d)



Data
25/10/1978
17/02/1981
20th March 2003
18/09/2000
02/07/2018
19/08/1950
32/05/1975
08/07/328
...

In una seconda fase possono essere trattati i valori delle date che hanno sollevato un'eccezione, in quanto qualche pattern matching non ha avuto successo. Tali valori sono stati marcati, e vanno trattati con regole di riscrittura ad hoc.

2.5.2 Esempi di trasformazioni di schema

I due esempi che seguono illustrano delle possibili sequenze di trasformazioni di schema, e le corrispondenti modifiche nella tabella. Queste sono operazioni tipiche in fase di consolidamento di sorgenti multiple.

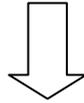
Esempio 2.

In seguito all'integrazione di dati provenienti da fonti eterogenee, è possibile avere una tabella come questa:

Steward, Bob	
Anna	Davis
Dole, Jerry	
Joan	Marsh

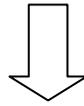
Tale situazione può essere affrontata come segue:

DIVIDE *colonna1* ...
 IF (MATCH “(.*),(.*)”) PUT INTO *colonna3*



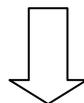
		Steward, Bob
Anna	Davis	
		Dole, Jerry
Joan	Marsh	

APPLY RWRULES ..ON *colonna3*
 << (.*),(.*) >> INTO \$2 \$1;



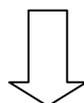
		Bob Steward
Anna	Davis	
		Jerry Dole
Joan	Marsh	

SPLIT *colonna3* ... AT (“ ”) ...



		Bob	Steward
Anna	Davis		
		Jerry	Dole
Joan	Marsh		

MERGE *colonna1* AND *colonna3* ...
 MERGE *colonna2* AND *colonna4* ...



Bob	Steward
Anna	Davis
Jerry	Dole
Joan	Marsh

Esempio 3.

Le trasformazioni di schema consentono anche di “normalizzare” i dati nel senso seguente:

Tabella di studenti

Nome	Esame A	Esame B
Ann	43	78
Bob	96	54

```
APPLY RWRULES...ON Esame A
<< (\d*)>> INTO “Esame A : $1”;
```

```
APPLY RWRULES...ON Esame B
<< (\d*)>> INTO “Esame B : $1”;
```

Nome	Esame A	Esame B
Ann	Esame A : 43	Esame B : 78
Bob	Esame A : 96	Esame B : 54

```
FOLD Esame A ON Esame B ...
```

Nome	Esame B
Ann	Esame A : 43
Ann	Esame B : 78
Bob	Esame A : 96
Bob	Esame B : 54

```
SPLIT Esame B ... AT ( “ : ” )
PUT INTO Esame AND Voto
```

```
APPLY RWRULES...ON Voto
<< :s(d*)>> INTO $1;
```

Tabella di esami

Nome	Esame	Voto
Ann	Esame A	43
Ann	Esame B	78
Bob	Esame A	96
Bob	Esame B	54

2.5.3 Rilevazione dei duplicati

Esempio 4.

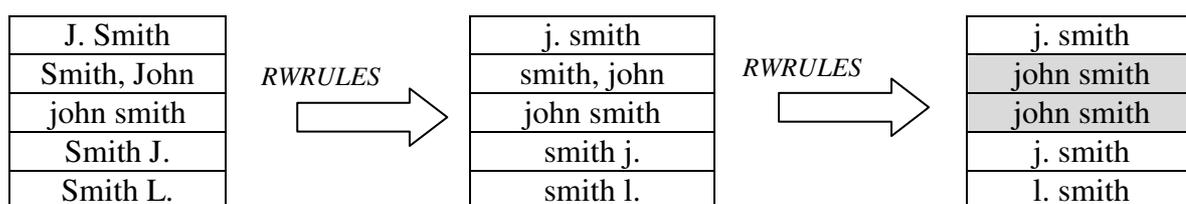
Problema:

Si vogliono individuare, tra i seguenti dati, i duplicati rispetto ai due attributi Nominativo e Indirizzo. (Si può ipotizzare che, quando il nome compare per esteso, esso precede sempre il cognome.)

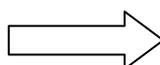
J. Smith	AddressA
Smith, John	AddressA
john smith	AddressA
Smith J.	AddressB
Smith L.	AddressB

Soluzione:

Innanzitutto, attraverso l'applicazione delle regole di riscrittura, si “normalizza” l'attributo *Nominativo* eliminando le maiuscole, e si anticipa il nome al cognome:



*MERGE DUPLICATES ...
ON (nominativo, indirizzo)*



j. smith	AddressA
john smith	AddressA
j. smith	AddressB
l. smith	AddressB

SPLIT nominativo AT “ ” PUT INTO cognome

NEWATTRIBUTE iniziale IS COPY OF nominativo

RWRULES ... ON iniziale <<([a-zA-Z])>> INTO \$1;

NEWATTRIBUTE abbreviazione IS COPY OF nominativo

RWRULES ... ON abbreviazione <<([a-zA-Z])(.)>> INTO IF (\$2=".") THEN 1 ELSE

0;

abbreviazione	iniziale	nominativo	cognome	indirizzo
1	j	j.	smith	AddressA
0	j	john	smith	AddressA
1	j	j.	smith	AddressB
1	l	l.	smith	AddressB

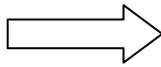
MARK DUPLICATES ... ON (iniziale, cognome, indirizzo) WITH CODE mioCodice

REMOVE ROW ... WHERE ((EXCEPTION mioCodice) AND (abbreviazione = 1))

MERGE nominativo AND cognome ...

REMOVE iniziale...

REMOVE abbreviazione ...



john smith	AddressA
j. smith	AddressB
l. smith	AddressB

2.5.4 Integrazione di sorgenti eterogenee

Esempio 5.

Problema:

Si vuole costruire un'unica tabella CLIENT a partire da tre sorgenti diverse, che ne integra tutto il contenuto informativo, senza ridondanze, e ripulita da anomalie interne alle sorgenti.

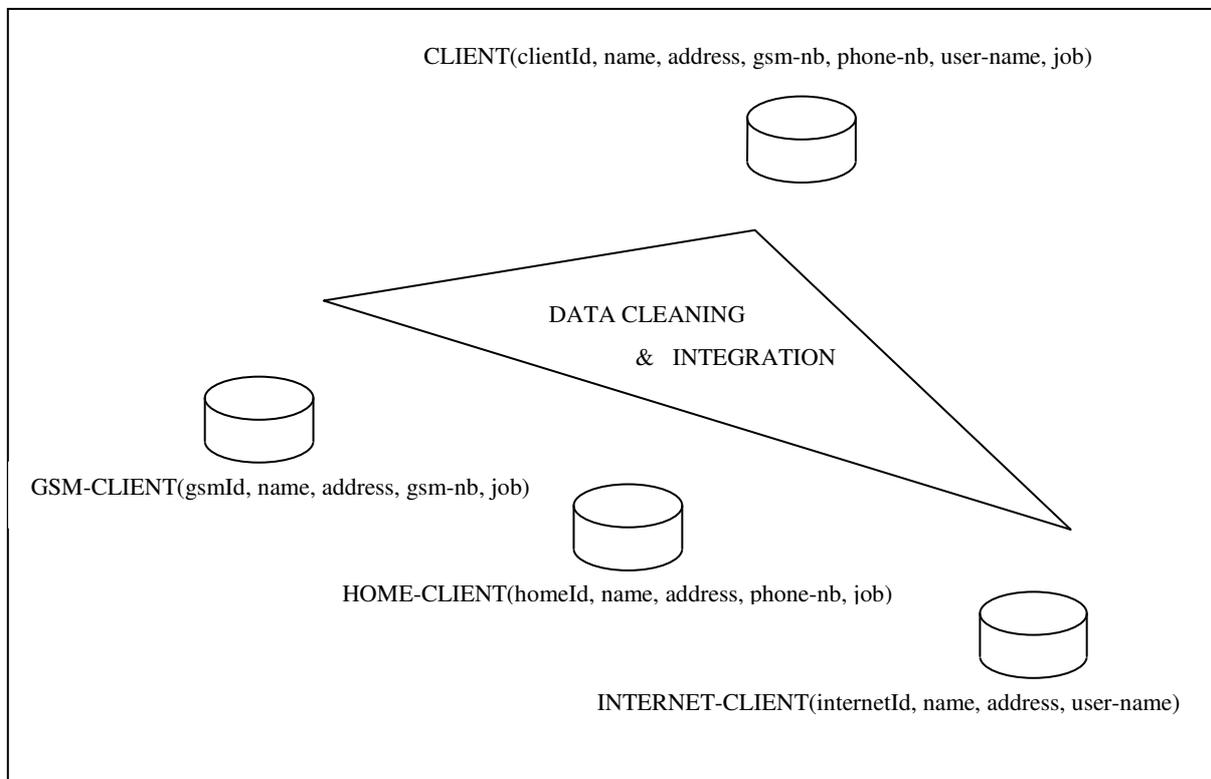


Figura 2.1: Integrazione di diverse sorgenti

In linea di massima si procede per tre fasi principali:

- Analisi dei dati
- Pulizia delle sorgenti
- Integrazione, e pulizia da ridondanze, inconsistenze, eterogeneità dei formati...

Osservazione:

I codici unici di identificazione delle sorgenti sono diversi ed indipendenti, ma è ragionevole supporre che, in generale, nome e indirizzo identifichino univocamente una persona (va comunque considerato che una persona può cambiare indirizzo, quanto i dati sono aggiornati, ...)

Analisi dei dati:

- a) si navigano ed esplorano le tre tabelle
- b) si effettuano statistiche e si utilizzano strumenti di visualizzazione
- c) eventuali discrepanze ed eccezioni sospette, venute a galla dai punti a) e b), si verificano con i costrutti *RWRULES (ruleM)* e *MARK*.

Pulizia delle sorgenti:

- a) si eliminano le anomalie interne: outliers, inconsistenze, ...
- b) si unificano i formati di nome e di indirizzo, utilizzando le regole di riscrittura *ruleS*, *ruleT*, similmente all'esempio 1., assieme ai costrutti *SPLIT*, *DIVIDE*, *NEWATTRIBUTE*, *MERGE*, *FOLD*, *REMOVE*
- c) si controlla, con la regola di marcatura *ruleM*, che tutti i nomi e indirizzi rispettino un certo pattern, altrimenti si trasformano ulteriormente le tuple marcate, perché non conformi al formato richiesto

Quando si può supporre che i formati siano uniformi, e che sono stati eliminati gli errori interni (vedi i punti 1-10 del paragrafo 2.1.2), si può procedere all'integrazione.

Integrazione:

- ogni tabella sorgente viene estesa aggiungendo dei nuovi attributi (*NEWATTRIBUTE*)
- si applica l'operatore SQL *UNION* alle tre tabelle, uniformi nello schema

ottenendo la tabella seguente:

Orig	new Id	idG	name -G	add-G	gsm-nb	job-G	IdH	Name -H	Add -H	phone -nb	Job -H	idI	Name -I	Add -I	user-name
A															
A															
A															
A															
A															
B															
B															
B															
B															
B															
C															
C															
C															
C															
C															

Si conclude consolidando la tabella sia in verticale che in orizzontale:

- MERGE(nameG, nameH, nameI) PUT INTO name
- MERGE(addG, addH, addI) PUT INTO address
- MERGE(jobH, jobG) PUT INTO job
- SELECT name, address, CONCAT(job), CONCAT(origine), CONCAT(idG), ...
GROUP BY name, address
- Assegna codice progressivo newId

N.B.: Dalla tabella integrata non vanno rimossi i codici di identificazione delle sorgenti originali, né l'informazione sulla provenienza di ogni tupla.

Osservazione:

Negli esempi, la sequenza di operatori DPL è applicata ad un'unica tabella in ingresso, la quale subisce le corrispondenti trasformazioni. I costrutti, per semplicità d'intuizione, sono riportati sempre in maniera sequenziale, ma rappresentano una composizione funzionale delle operazioni, applicata alla tabella d'ingresso.

Osservazioni sull'utilizzo dei costrutti:

1.) Si noti che è possibile effettuare la trasformazione inversa rispetto a FOLD, illustrata sotto, tramite l'utilizzo dei seguenti costrutti:

- `SELECT Nome, CONCAT(Esame)`
`GROUP BY Nome`
- `SPLIT Esame AT "Esame B"`
`PUT INTO Esame A AND Esame B`

Nome	Esame
Ann	Esame A : 43
Ann	Esame B : 78
Bob	Esame A : 96
Bob	Esame B : 54



Nome	Esame A	Esame B
Ann	Esame A : 43	Esame B : 78
Bob	Esame A : 96	Esame B : 54

2.) È possibile marcare le tuple che non hanno raggiunto una soglia minima di frequenza, rispetto ad un certo attributo, individuando così delle possibili anomalie. A tale scopo si applica prima l'ordinamento per frequenza, e successivamente l'operatore di marcatura, specificando come condizione il mancato raggiungimento della soglia da parte dell'*attributo di frequenza*.

- 3.) Per aggiungere un nuovo attributo ad una tabella, il quale è costituito da una elaborazione della sequenza di caratteri di un altro attributo, è necessario creare prima una copia di tale attributo, e solo successivamente si può sfruttare il meccanismo del pattern matching attraverso le regole di riscrittura.
- 4.) Si noti come l'operazione del punto precedente può essere interessante in fase di integrazione: attraverso il pattern matching si possono “estrarre” da un attributo le componenti strutturali che si ritengono di rilievo, per realizzare successivamente una giunzione *per similitudine*. Un altro interessante esempio: si possono marcare le tuple che costituiscono dei possibili duplicati, perché soddisfano lo stesso *pattern*.
- 5.) Attraverso le regole di riscrittura *ruleS* e *ruleT*, il sistema è esteso con la possibilità di verificare delle condizioni di “similitudine”.

2.6 Sintassi

Dopo avere introdotto informalmente i costrutti di trasformazione del linguaggio proposto, anche tramite qualche esempio, viene di seguito data la grammatica che ne definisce la sintassi. Si tratta, per la precisione, di un'estensione alla sintassi MQL, il linguaggio di interrogazione data mining sviluppato per il sistema KDDML. La componente DPL, nella grammatica, è evidenziata in grassetto. Si noti, che ogni operatore DPL consiste in una trasformazione di una tabella relazionale, e può perciò essere visto come un'interrogazione che restituisce una tale tabella. Si anticipa inoltre che, per realizzare il meccanismo di marcatura, in fase di preparazione dei dati si farà uso di *tabelle estese*, rappresentate dall'oggetto sintattico *PPTable*.

La trattazione semantica sarà oggetto del prossimo capitolo.

Sintassi estesa di MQL

```

MQL ::= begin query <QUERY> end query
<QUERY> ::= let <Q> in | <Q1>
<Q> ::= <Q1> <Q2>
<Q1> ::= <NAME> = { <RuleQuery>|<TreeQuery>|<ClusterQuery>|
                    <TableQuery> | <PPTableQuery> }
<Q2> ::= ;<Q>| ε
<RuleQuery> ::= CreateRules { <T> | (<TableQuery>) }
<AlgRules>|
                CreateGenRules <G> { <T> | (<TableQuery>) }
<AlgGen> |
                filter { <R> | (<RuleQuery>) } if <cond> |
                PreservedRule { <R> | (<RuleQuery>) } <Rg> <G>
<N>
<TreeQuery> ::= CreateTree { <T> | (<TableQuery>) } <AlgTree> |
                Exec <Op>
<ClusterQuery> ::= CreateCluster {<T>| (<TableQuery>) }
<AlgClus>
<TableQuery> ::= RdA2Table { <R> | (<RuleQuery>) } |
                take <Elem> by {<T>| (<TableQuery>) } [ if
<cond>] |
                Use { <Tree> | (<TreeQuery>) } With {<T>|
(<TableQuery>) }|

```

```

Rule { <R> | (<RuleQuery>) } Support {<T> |
(<TableQuery>) } |

Rule{ <R>| (<RuleQuery>) } Exception{<T> |
(<TableQuery>) }|

Cluster { <C> | (<ClusterQuery>) } Number <N>
|

Max { <C> | (<ClusterQuery>) } |

Misclassified {<T> | (<TableQuery>) }|

ExtractAll { <C> | (<ClusterQuery>) }|

PPTable2Table { <P> | (<PPTaleQuery>) }
<PPTableQuery> ::= <RewritingQuery> [ <restr> ] | <MarkingQuery> [ <restr> ] |
<DividingQuery> | <SplittingQuery> | <MergingQuery> |
<RenameQuery> | <RemoveQuery> | <NewAttributeQuery> |
<FoldingQuery> | <Discretization> | <Normalization> [ <restr> ] |
<Sampling> | <SortingByFrequency> | <MarkDuplicates> |
<MergeDuplicates> | <SQLQuery> | <RemoveRow> <restr> |
Table2PPTable {<T>| (<TableQuery>) }

/* Costrutto di riscrittura */
<RewritingQuery> ::= APPLY RWRULES IN <PPTable> TO <Attribute>
BEGIN
    <Rewriting_rules>
END
<Rewriting_rules> ::= <Rule> ; | <Rule> ; <Rewriting_rules>
<Rule> ::= <RuleS> | <RuleM> | <RuleT>
<RuleS> ::= <Pattern> INTO <Statement> [ EXCEPTION [<Code>] ]
<Pattern> ::= « <ER>1 »
<Statement> ::= <Substitution>
| IF (<rw_cond>) THEN <Statement>

```

¹ ER rappresenta un'espressione regolare, come quelle utilizzate nel linguaggio Perl.

```

| IF (<rw_cond >) THEN <Statement> ELSE <Statement>
<Substitution> ::= 'stringa di sostituzione contenente variabili di raggruppamento $i e
caratteri'
<rw_cond > ::= <rw_term> <compare_op > <rw_term >
| (<rw_cond >) AND (<rw_cond >)
| (<rw_cond >) OR (<rw_cond >)
| NOT <rw_cond >
<rw_term > ::= 'una variabile di raggruppamento $i oppure una sequenza di caratteri'
<RuleM> ::= <Pattern> EXCEPTION [<Code >]
<RuleT> ::= EVERY <Pattern> INTO <Substitution> [ EXCEPTION [<Code >] ]

```

/ Costrutto di marcatura */*

```

<MarkingQuery> ::= MARK <Attribute> IN <PPTable> IF (<mark_cond >)
[ WITH CODE <Code> ]
<mark_cond > ::= <mark_term> <compare_op > <mark_term>
| (<mark_cond >) AND (<mark_cond >)
| (<mark_cond >) OR (<mark_cond >)
| NOT (<mark_cond >)
<mark_term> ::= <Expression>
| <term>

```

/ Costrutto di divisione */*

```

<DividingQuery> ::= DIVIDE <Attribute> IN <PPTable> IF (<div_cond >)
PUT INTO <NewAttribute>
<div_cond > ::= <compare_op> <div_term> , <div_term >
| MATCH <ER>
| NOT <div_cond >
| (<div_cond >) (AND|OR) (<div_cond >)
<div_term > ::= SOURCE | <constant_term>

```

/ Costrutto di split */*

```

<SplittingQuery> ::= SPLIT <Attribute> IN <PPTable> AT (<ER>)
PUT INTO <NewAttribute> AND <NewAttribute>

```

[EXCEPTION [<Code>]]

/ Costrutto di fusione */*

**<MergingQuery> ::= MERGE <Attribute> AND <Attribute> IN <PPTable>
PUT INTO <NewAttribute>**

/ Costrutto di rinominazione */*

<RenameQuery> ::= RENAME <Attribute> IN <PPTable> AS <NewAttribute>

/ Costrutto di rimozione */*

<RemoveQuery> ::= REMOVE <Attribute> IN <PPTable>

/ Costrutto di creazione di un nuovo attributo */*

**<NewAttributeQuery> ::= NEWATTRIBUTE <NAME> IN <PPTable> [IS <newAtt_exp>]
< newAtt_exp > ::= <Expression> | COPY OF <Attribute>**

/ Costrutto di ripiegamento */*

<FoldingQuery> ::= FOLD <Attribute> ON <Attribute> IN <PPTable>

/ Costrutti di discretizzazione numerica, categorica e supervisionata */*

<Discretization> ::= <Discret_num> | <Discret_categ> | <Discret_supervis>

**<Discret_num> ::= DISCRETIZE (EQUIWIDTH | EQUIDEPH) <Attribute> IN <PPTable>
WITH (WIDTH <N> | DEPTH <N> | <N> INTERVALS)
SMOOTH BY (MEAN | MEDIAN | INF | SUP | <Enumeration>)
[EXCEPTION [<Code>]]**

**<Discret_categ> ::= (SPECIFY ORDERING {<T>| (<TableQuery>) } ON <Discret_num>)
| (SPECIFY HIERARCHY {<T>| (<TableQuery>) } ON
DISCRETIZE <Attribute> IN <PPTable> EXCEPTION [<Code>])**

**<Discret_supervis> ::= DISCRETIZE <Attribute> IN <PPTable>
SUPERVISED WITH <Supervision_method>
USING CLASS <Attribute> STOP AT <R_e>
SMOOTH BY (MEAN | MEDIAN | INF | SUP | <Enumeration>)
[EXCEPTION [<Code>]]**

< Supervision_method > ::= CHI2 | ENTROPY

<Enumeration> ::= { <NAME> <Enum> }

<Enum> ::= , <NAME> <Enum> | ε

/ Costrutto di normalizzazione */*

**<Normalization> ::= NORMALIZE_<norm_method> (<attribute_seq>) IN <PPTable>
[EXCEPTION [<Code>]]**

**< norm_method > ::= (MIN_MAX <R_e> <R_e>) | Z_SCORE | DECIMAL |
(MIN_MAX_L <R_e> <R_e> <R_e> <R_e>) | (Z_SCORE_L <R_e> <R_e>)**

<attribute_seq> ::= (<Attribute> , <attribute_seq>) | <Attribute>

/ Costrutto di campionamento */*

<Sampling> ::= SAMPLE <PPTable> <sample_method> USE <sample_algorithm>

<sample_method> ::= SIMPLE

| BY CLUSTER <Attribute>

| STRATIFIED ON <Attribute>

<sample_algorithm> ::= (SRSWOR | SRSWR) WITH (<N> ITEMS | INDEX <R_e>)

/ Costrutto di ordinamento per frequenza */*

<SortingByFrequency> ::= SORT <PPTable> BY FREQUENCY ON <Attribute>

/ Costrutto di marcatura dei duplicati */*

**<MarkDuplicates> ::= MARK DUPLICATES IN <PPTable> ON (<attribute_seq>)
[WITH CODE <Code>]**

/ Costrutto di fusione dei duplicati */*

**<MergeDuplicates> ::= MERGE DUPLICATES IN <PPTable> ON (<attribute_seq>)
[WITH CODE <Code>]**

/ Operatori SQL */*

<SQLQuery> ::= un'espressione SQL che crea una Vista Relazionale

/ Operatore di rimozione di tuple */*

<RemoveRow> ::= REMOVE ROW IN <PPTable>

/ Clausola di restrizione sulle tuple */*

<restr> ::= WHERE <restriction_cond>

<restriction_cond> ::= <term> <compare_op> <term>

| NOT <restriction_cond>

| (<restriction_cond>) (AND|OR) (<restriction_cond>)

| EXCEPTION [<Code>] [ON <Attribute>]

<Expression> ::= (<term_operation> <term_seq>)

<term_operation> ::= qualunque operazione su stringhe o numeri

<term_seq> ::= <term> , <term_seq> | <term>

<term> ::= ATTRIBUTE <Attribute>

| <Expression>

| <constant_term> /* qualunque costante numerica o di tipo stringa

<compare_op> ::= una qualunque operazione di confronto θ tra stringhe oppure numeri

<constant_term> ::= string | real | integer

<attribute_seq> ::= (<Attribute> , <attribute_seq>) | <Attribute>

<AlgRules> ::= <Apriori> <Re> <Re> <N>

<AlgGen> ::= <GenApriori> <Re> <Re> <N>

<AlgTree> ::= C4.5 <Re> <N> | ID3

<AlgClus> ::= EM <N> <N> | <NAME>

<R_g> ::= <R> | CreateGenRules <G> {<T> | (<TableQuery>) }

<AlgGen>

<Op> ::= And ({ <Tree> | (<TreeQuery>) }, { <Tree> | (<TreeQuery>)

} |

Or ({ <Tree> | (<TreeQuery>) }, { <Tree> | (<TreeQuery>) })

|

Comitato({ <Tree> | (<TreeQuery>) }, { <Tree> | (<TreeQuery>) })

<PPTable> ::= <P> | <PPTableQuery>

<T> ::= Tabella

<P> ::= Tabella Estesa

<Attribute> ::= Attributo

<Tree> ::= Albero

<R> ::= RdA

<C> ::= Clusters

<G> ::= Gerarchia

<R_e> ::= real

<N> ::= integer

** ::= true | false**

<NAME> ::= string

<Elem> ::= <NAME> <A> | all

<A> ::= ε | ,<NAME> <A>

<NewAttribute> ::= string

<Code> ::= string

<ER> ::= un'espressione regolare che definisce il 'pattern' di una sequenza di caratteri

La grammatica che segue definisce le *espressioni regolari* valide nel formalismo DPL, e che servono per descrivere uno schema di costruzione (*pattern*) di una stringa.

<ER>	::=	<unione> <ER-semplce>
<unione>	::=	<ER> " " <ER-semplce>
<ER-semplce>	::=	<concatenazione> <ER-base>
<concatenazione>	::=	<ER-semplce> <ER-base>
<ER-base>	::=	<asterisco> <più> <parentesi-graffe> <ER-elementare>
<asterisco>	::=	<ER-elementare> "*" "
<più>	::=	<ER-elementare> "+" "
<parentesi-graffe>	::=	<ER-elementare> "{" intero positivo "}" <ER-elementare> "{" intero positivo ", }" <ER-elementare> "{" intero positivo ", " intero positivo "}" "
<ER-elementare>	::=	<gruppo> <qualunque> <fine-stringa> <carattere> <insieme>
<gruppo>	::=	" (" <ER> ") "
<qualunque>	::=	". "
<fine-stringa>	::=	"\$"
<carattere>	::=	qualunque non metacarattere "\" <metacarattere> "\" <metacarattere>
<insieme>	::=	<insieme-positivo> <insieme-negativo>
<insieme-positivo>	::=	"[" <elementi-di-insieme> "]" "
<insieme-negativo>	::=	"[^" <elementi-di-insieme> "]" "
<elementi-di-insieme>	::=	<elemento-di-insieme> <elemento-di-insieme> <elementi-di-insieme>
<elemento-di-insieme>	::=	<intervallo> <carattere>
<intervallo>	::=	<carattere> "-" <carattere>
<metacarattere>	::=	d D w W s S n r t b 0 xHH

Capitolo 3

Semantica

Dopo aver introdotto il linguaggio attraverso dei semplici esempi di utilizzo, verrà di seguito data una descrizione più dettagliata e formale della sua semantica. L'approccio che si è scelto è volutamente uniforme a quello adottato nella definizione semantica del linguaggio per interrogazioni di data mining MQL, di cui il presente lavoro vuole essere un'integrazione

3.1 La semantica del linguaggio

Le espressioni principali del nostro linguaggio rappresentano delle tabelle relazionali, estese con particolari informazioni in modo da riuscire a tenere traccia della storia delle modifiche apportate a tali tabelle, e contemporaneamente per realizzare il meccanismo di marcatura delle eccezioni. Nello scegliere una struttura di astrazione per rappresentare le espressioni, si è deciso di adottare la *sequenza*, perché è sembrata essere la struttura di più alto livello che fosse in grado di modellare le espressioni del linguaggio. Oltre a questo, la sequenza costituisce lo stesso modello di astrazione che è stato adottato nella definizione del linguaggio per interrogazioni di data mining MQL, del quale DPL vuole essere un'estensione¹. Si vorrebbe infatti possibilmente integrare i due linguaggi, per permettere in un unico ambiente di effettuare sia la preparazione dei dati che le interrogazioni di data mining. Si rivelerebbe in effetti interessante poter estrarre modelli di conoscenza, ad esempio dei predittori, per utilizzarli nella ricerca di *outliers* in fase di preparazione. L'alternanza di fasi di preparazione a fasi di data

¹ Il linguaggio MQL è stato sviluppato nella tesi [Tesi2]

mining non sarebbe dunque una pura utilità pratica per l'utente, ma rappresenterebbe un incremento di potenzialità e di espressività importante.

La semantica del linguaggio che daremo di seguito è una semantica operativa e definita nel linguaggio introdotto in [Tesi2], le cui operazioni primitive sono gli operatori su sequenze. Considereremo poi ogni costrutto di trasformazione del linguaggio come un operatore derivato dagli operatori elementari sulle sequenze. Per le definizioni di tali operatori elementari si rimanda all'appendice A e per una descrizione più dettagliata si rimanda a [Tesi2].

3.2 Le tabelle estese

L'espressione principale del linguaggio DPL è la *tabella estesa*, che rappresenta fondamentalmente una tabella relazionale. Rispetto alla tabella relazionale definita nel linguaggio MQL, essa ne forma un'estensione che contiene delle metainformazioni riguardanti la storia delle trasformazioni subite dalla tabella, e contemporaneamente realizza il meccanismo di marcatura delle eccezioni. Da un lato la marcatura sarà di utilità pratica in fase di analisi dei dati, ma sarà anche in grado di esprimere le condizioni di iterazione del flusso di controllo.

Diamo di seguito una descrizione più dettagliata delle tabelle estese e del *tipo* attraverso cui sono rappresentate in DPL.

3.2.1 Il tipo 'TabEstesa'

L'estensione della tabella è realizzata aggiungendo un campo informativo per ogni attributo, ossia raddoppiando il numero di colonne della tabella. Possiamo allora immaginarci che la tabella estesa abbia la seguente forma:

Attr1	Attr2	Attr3	Attr4	Attr5					
			valore					metadati	

Figura 3.1: Una tabella estesa

Preso una tupla, ogni attributo ha associato un valore corrente, e dei metadati che tengono traccia della storia delle trasformazioni subite. Queste ultime si trovano nel campo informativo corrispondente. Se vediamo una tupla come una sequenza di n valori, dove n è il numero di attributi dello schema relazionale, allora possiamo realizzare una tupla estesa come una sequenza lunga $2*n$, dove in posizione i (i compreso tra 1 e n) si trova il valore attuale di un certo attributo e in posizione $i+n$ si trovano i metadati corrispondenti (vedi fig. 4.1).

È previsto che i metadati siano memorizzati su richiesta esplicita, facendo seguire ad un costrutto di trasformazione il tag *Exception*, con un eventuale codice di marcatura. In questo caso se la trasformazione dovesse sollevare un'eccezione, verrà *marcato* il valore dell'attributo memorizzando nel campo informativo associato:

- a) il tipo di trasformazione in questione, assieme ai parametri attuali,
- b) il valore originale (e non modificato) dell'attributo, e
- c) se specificato, il *codice di marcatura*.

Il valore dei campi dell'estensione formeranno perciò delle sequenze di triple.

Ricordiamo come era stata rappresentata una tabella, in termini di sequenze, nel linguaggio MQL:

Tabella
@Attributi
@NumAttr
@Valori(@Sequenza)#

Si può rappresentare la tabella estesa descritta sopra, in modo uniforme alla tabella tradizionale, nel modo seguente:

TabEstesa
@Attributi
@NumAttr
@Valori(@SequenzaEstesa)#¹

¹ Una possibile tabella è: (((matricola, pin),2,((207239, 34712),(213242, 24928),(203481, 58321))));

una tabella estesa corrispondente è:

(((matricola, pin),2,((207239, 34712, null, <metadati associati al pin 34712>),(213242, 24928, null, null),(203481, 58321, null, null))));

Per un approfondimento del modello di astrazione delle *sequenze*, adottato nella definizione semantica di MQL e DPL, si rimanda a [Tesi2].

dove le sequenze corrispondenti a @SequenzaEstesa rappresentano le “tuple estese”.

3.2.2 Il dominio delle *tabelle estese*

Il linguaggio DPL è costituito principalmente da costrutti che rappresentano delle trasformazioni di tabelle. Dunque gli ingressi e le uscite di tali trasformazioni sono oggetti di tipo *TabEstesa*. Su di essi si può definire una relazione di ordinamento parziale ben fondato, in modo da poter successivamente adottare una definizione per casi e per ricorsione ben fondata nel dare la semantica operativa del linguaggio.

D’ora in poi utilizzeremo la notazione **T[Valori]** per rappresentare la sottosequenza corrispondente a @Valori nell’oggetto T di tipo *TabEstesa*; esso contiene le tuple della tabella estesa T. Analogamente indicheremo con **T[Attributi]** e con **T[NumAttr]** i valori attuali dell’elemento @Attributi e dell’elemento @NumAttr nell’oggetto rappresentato da T.

Definizione 1 : (\leq_I)

Indichiamo con il simbolo \leq_I la relazione binaria sul dominio delle tabelle estese, in modo che posto

T1, T2 di tipo *TabEstesa*,

T1[Attributi] = T2[Attributi]

T1[NumAttr] = T2[NumAttr]

allora

T1 \leq_I T2 se e solo se **T1[Valori]** forma una sottosequenza finale di **T2[Valori]**.

□

Ad esempio se

$$\mathbf{T1[Valori]} = (t1, t2, t3)$$

$$\mathbf{T2[Valori]} = (t2, t3)$$

$$\mathbf{T3[Valori]} = (t1, t2)$$

$$\mathbf{T4[Valori]} = (t3, t2)$$

allora

$$\mathbf{T2} \leq_I \mathbf{T1}$$

$$\mathbf{T3} \not\leq_I \mathbf{T1}$$

$$\mathbf{T4} \not\leq_I \mathbf{T2}.$$

Si noti che tale relazione è un ordinamento parziale, completo e ben fondato.

Definizione 2 : (\equiv_E)

Indichiamo con il simbolo \equiv_E la relazione binaria sul dominio delle tabelle estese, in modo che se

$\mathbf{T1}, \mathbf{T2}$ di tipo *TabEstesa*,

$$\mathbf{T1[Attributi]} = \mathbf{T2[Attributi]}$$

$$\mathbf{T1[NumAttr]} = \mathbf{T2[NumAttr]}$$

allora

$\mathbf{T1} \equiv_E \mathbf{T2}$ se e solo se $\mathbf{T1[Valori]}$ è identicamente uguale a $\mathbf{T2[Valori]}$ per qualche riordinamento della sua sequenza.

□

Ad esempio $\mathbf{T2} \equiv_E \mathbf{T4}$. \equiv_E indica l'uguaglianza su insiemi, e forma una relazione di equivalenza.

Definizione 3 : (\leq_E)

Indichiamo con il simbolo \leq_E la relazione binaria sul dominio delle tabelle estese, in modo che se

T1, T2 di tipo *TabEstesa*,

T1[Attributi] = T2[Attributi]

T1[NumAttr] = T2[NumAttr]

allora

T1 \leq_E T2

se e solo se

esiste **T'** di tipo *TabEstesa* t.c. **T' \leq_I T2** e **T1 \equiv_E T'**.

□

Anche questa relazione forma un ordinamento parziale, completo e ben fondato. In riferimento all'esempio di sopra, vale che

T2 \leq_E T1

T3 \leq_E T1

T4 \leq_E T1

T3 $\not\leq_E$ T2.

Definizione 4 : (\cup_T)

L'unione sulle tabelle

$\cup_T : \text{TabEstesa} \times \text{TabEstesa} \rightarrow \text{TabEstesa}$

è definita come la concatenazione degli elementi presenti nella sequenza dell'elemento *Valori* di un oggetto *TabEstesa*, e di conseguenza le due tabelle devono avere lo stesso schema. La definizione dell'unione in termini di operatori su sequenze è la seguente:

se **T1**, **T2** di tipo *TabEstesa*,

$$\mathbf{T1[Attributi] = T2[Attributi]}$$

$$\mathbf{T1[NumAttr] = T2[NumAttr]}$$

allora

$$\mathbf{T1 \cup_T T2 \equiv}$$

$$\cup_T(\mathbf{T1}, \mathbf{T2}) \text{ is}$$

$$\gamma(\text{TabEstesa}, ((\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, \mathbf{T1}), 1),$$

$$\delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, \mathbf{T1}), 1),$$

$$\alpha(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, \mathbf{T1}), 1),$$

$$\delta(\epsilon(\text{TabEstesa}, \text{Valori}, \mathbf{T2}), 1)))$$

□

3.3 Nozioni di base per l'implementazione

Inizialmente verranno introdotte alcune utili operazioni sulle sequenze, che ricorrono frequentemente in questo capitolo. Si è deciso di raccogliere e riportare tali definizioni nella parte iniziale perché costituiscono un supporto per il lavoro successivo. Per questo motivo il paragrafo 3.3.1 deve essere utilizzato come un riferimento durante la lettura dei paragrafi che seguono, piuttosto che come una base su cui poggia la definizione semantica del linguaggio. La lettura del primo paragrafo può dunque essere rimandata senza pregiudicare la comprensione del lavoro successivo.

3.3.1 Operazioni derivate frequenti

Nell'implementazione del linguaggio si è cercato di adottare un approccio strutturato, per rendere il più possibile uniformi le realizzazioni delle varie trasformazioni, e per mettere in evidenza le caratteristiche comuni. A questo scopo introduciamo ora alcune operazioni sulle tabelle estese, che costituiscono delle operazioni derivate sul dominio delle sequenze, e che saranno utilizzate frequentemente in questo capitolo.

Viene aggiunta la tupla estesa 'TP' in cima alla tabella relazionale estesa 'TE', ossia viene concatenata 'TP' in testa alla sequenza @Valori nell'oggetto TabEstesa 'TE'.

Aggiungi(TP, TE) is

$$\gamma(\text{TabEstesa}, ((\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, \text{TE}), 1), \\ \delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, \text{TE}), 1), \\ \alpha((\text{TP}), \delta(\epsilon(\text{TabEstesa}, \text{Valori}, \text{TE}), 1)))))$$

Elimina il primo elemento dalla sequenza 'S'

EliminaIlPrimo(S) is

$$\mu(S, \delta(S, 1))$$

Operatore derivato: elimina l'elemento in posizione P dalla sequenza S

$$\lambda(S, P) \equiv \text{Elimina}(S, P)$$

dove

Elimina(S,P) is

if $(P \leq 0)$ then S

else if $(P == 1)$ then EliminaIlPrimo(S)

else $\alpha((\delta(S, 1)), \text{Elimina}(P-1, \text{EliminaIlPrimo}(S)))$

Operatore derivato: concatena il valore V in fondo all'elemento in posizione P della sequenza S

$$\eta(V, S, P) \equiv \text{Concatena}(V, S, P)$$

dove

Concatena(V,S,P) is

if $(P \leq 0)$ then S

else if $(P == 1)$ then $\alpha(\delta(S, 1) @^1 V)$, EliminaIlPrimo(S)

else $\alpha(\delta(S, 1), \text{Concatena}(V, P-1, \text{EliminaIlPrimo}(S)))$

Operatore derivato: sostituisce il valore V all'elemento in posizione P della sequenza S

$$\phi(V, S, P) \equiv \text{Sostituisci}(V, S, P)$$

dove

Sostituisci (V,S,P) is

if $(P \leq 0)$ then S

else if $(P == 1)$ then $\alpha(V)$, EliminaIlPrimo(S)

else $\alpha(\delta(S, 1), \text{Sostituisci}(V, \text{EliminaIlPrimo}(S), P-1))$

Operatore derivato: inserisce il valore V in posizione P della sequenza S

$$\zeta(V, S, P) \equiv \text{Inserisci}(V, S, P)$$

dove

Inserisci(V,S,P) is

if $(P \leq 0)$ then S

else if $(P == 1)$ then $\alpha(V)$, S

else $\alpha(\delta(S, 1), \text{Inserisci}(V, \text{EliminaIlPrimo}(S), P-1))$

¹ Con il simbolo @ è espressa la concatenazione dei valori che costituiscono gli elementi della sequenza. In questo caso si tratta dei valori che descrivono le eccezioni verificatesi durante le trasformazioni; nel caso più semplice potrebbero essere delle stringhe.

RiduciTabella elimina la prima tupla della tabella estesa T , ossia il primo elemento della sottosequenza @Valori

RiduciTabella(T) is

$$\gamma(\text{TabEstesa}, ((\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, T), 1), \\ \delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, T), 1), \\ \lambda(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, T), 1), 1) \quad)) \quad))$$

NuovaTupla sostituisce nella tupla TP l'attributo in posizione P con il valore NV , e nel caso in cui l'eccezione E non fosse null, la aggiunge, concatenandola al valore corrente, in posizione $P+NA$

NuovaTupla(TP, NV, P, E, NA) is

$$\text{if } (E == \text{null}) \text{ then } \varphi(NV, TP, P) \\ \text{else } \eta(E, \varphi(NV, TP, P), P+NA)$$

Fschema_AggiungiColonna estende lo schema relazionale della tabella estesa T con il nuovo attributo A e con il relativo campo di estensione per la marcatura; lascia inalterata la sequenza Valori della tabella T

Fschema_AggiungiColonna(T, A) is

$$\gamma(\text{TabEstesa}, ((\zeta(\text{Est}_+A, \\ \zeta(A, \\ \delta(\epsilon(\text{TabEstesa}, \text{Attributi}, T), 1), \\ \delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, T), 1)+1), \\ 2*(\delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, T), 1)+1)), \\ \delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, T), 1)+1, \\ \delta(\epsilon(\text{TabEstesa}, \text{Valori}, T), 1) \quad)) \quad))$$

Fschema_EliminaColonna opera solamente a livello di schema relazionale, eliminando la colonna relativa all'attributo *A*, ed anche la corrispondente parte di estensione della tabella, utile per il meccanismo di marcatura; essa restituisce una terna, formata dalla tabella modificata nello schema, dalla posizione dell'attributo rimosso, e dal numero di attributi presenti in seguito alla rimozione

Fschema_EliminaColonna(T, A) is

$$\begin{aligned}
 & (\gamma(\text{TabEstesa}, ((\lambda(\lambda(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, T), 1), \\
 & \quad \tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, T), 1), A) \\
 & \quad + \delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, T), 1)), \\
 & \quad \tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, T), 1), A)), \\
 & \quad \delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, T), 1) - 1, \\
 & \quad \delta(\epsilon(\text{TabEstesa}, \text{Valori}, T), 1) \quad)) \quad), \\
 & \tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, T), 1), A)), \\
 & \delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, T), 1) - 1)
 \end{aligned}$$

ValutazioneSemantica è una funzione ricorsiva, la quale calcola il valore dell'espressione *Espr*, rappresentata dal suo albero sintattico, ossia dal suo schema di costruzione

ValutazioneSemantica(TP, Tab, Espr) is

case (Figli(Espr)) of

{

("ATTRIBUTE " <attributo>) :

$$\delta(\text{TP}, \tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, \text{Tab}), 1),$$

$$\text{Figlio}(\text{Figlio}(\text{Espr}, 2), 1) \quad))$$

((" <Operazione> <SeqOperandi> ")) :

$$\text{ValutazioneSemantica}(\text{TP}, \text{Tab}, \text{Figlio}(\text{Espr}, 2))$$

$$(\text{ValutazioneSemantica}(\text{TP}, \text{Tab}, \text{Figlio}(\text{Espr}, 3)) \quad)$$

(θ): // * θ può indicare una qualunque operazione su numeri o stringhe
 θ_{sem}

(<Operando>):

ValutazioneSemantica (TP, Tab, Figlio(Espr,1))

(<Operando> “,” < SeqOperandi>):

(ValutazioneSemantica (TP, Tab, Figlio(Espr,1)) ,

ValutazioneSemantica (TP, Tab, Figlio(Espr,3)))

(c): //* *c* può indicare una qualunque costante numerica o di tipo stringa

C_{sem}

}

Tra le primitive del linguaggio sono anche previste le conversioni di oggetti dal tipo *Tabella* a *TabEstesa*, e viceversa:

Table2PPTable(T) is

γ (TabEstesa, ((δ (ϵ (Tabella,Attributi, T),1),

δ (ϵ (Tabella,NumAttr, T),1),

EstendiValori(δ (ϵ (Tabella,Valori, T),1), T)))

EstendiValori estende ogni tupla, contenuta nella sequenza *SeqValori*, con i campi di estensione; restituisce una sequenza di tuple estese, ossia di oggetti di tipo ‘*SequenzaEstesa*’

EstendiValori(SeqValori, Tab) is

if (δ (SeqValori, 1) == null)

then null

else α (EstendiTupla(δ (SeqValori, 1),

δ (ϵ (Tabella,NumAttr, Tab),1)),

EstendiValori(λ (SeqValori, 1), Tab))

EstendiTupla estende la tupla con un campo di estensione per ogni attributo; la funzione è ricorsiva sul numero di attributi ancora da estendere, inizialmente NA

EstendiTupla(Tupla, NA) is

```

if (NA ≤ 0) then Tupla
    else EstendiTupla( α(Tupla, (null)), NA-1)

```

La trasformazione da una tabella estesa ad una tabella tradizionale è definita nel seguente modo:

PPTable2Table(T) is

```

γ( Tabella, (( δ(ε( TabEstesa,Attributi, T),1),
              δ(ε( TabEstesa,NumAttr, T),1),
              EstraiValori(δ(ε( TabEstesa,Valori, T),1), T) )) )

```

EstraiValori estrae da ogni tupla estesa, contenuta nella sequenza SeqValoriEstesi, i campi di estensione; restituisce una sequenza di tuple, ossia di oggetti di tipo 'Sequenza'

EstraiValori(SeqValoriEstesi, Tab) is

```

if (δ(SeqValoriEstesi, 1) == null)
    then null
    else α ( EstraiTupla( δ(SeqValoriEstesi, 1),
                        δ(ε( TabEstesa,NumAttr, Tab),1)
                        δ(ε( TabEstesa,NumAttr, Tab),1) ),
            EstraiValori( λ(SeqValoriEstesi, 1), Tab ) )

```

EstraiTupla, per ogni attributo, estrae il campo di estensione dalla tupla estesa; la funzione è ricorsiva sul numero di attributi che possiedono ancora l'estensione, inizialmente N

EstraiTupla(TuplaEstesa, N, NA) is
 if $(N \leq 0)$ then TuplaEstesa
 else EstraiTupla($\lambda(\text{TuplaEstesa}, \text{NA}+\text{N}), \text{N}-1, \text{NA}$)

3.3.2 Le trasformazioni “locali”

Durante il presente lavoro si è riconosciuta l'importanza di un metodo strutturato nella definizione formale del linguaggio. A questo scopo viene introdotta ora una classe particolare di trasformazioni di tabelle estese, le quali hanno in comune delle importanti proprietà, le quali verranno trattate nel capitolo successivo. Tuttavia è importante individuare le trasformazioni appartenenti a questa classe sin dall'inizio, per definirne la semantica nel modo più uniforme possibile, e mettendo in evidenza l'omogeneità della struttura. In questo modo risulterà più leggibile ed intuitiva l'implementazione delle trasformazioni, e le proprietà semantiche potranno successivamente essere definite con valenza su una intera collezione di trasformazioni, e dimostrate ad un livello di astrazione superiore, con notevoli vantaggi di comprensibilità.

In figura 3.2 è rappresentato lo schema logico di una trasformazione f applicata alla tabella estesa TE. Chiameremo d'ora in poi **trasformazione locale** una trasformazione che può essere implementata secondo lo schema di fig. 3.2; la sua caratteristica fondamentale è che ogni tupla della tabella, soggetta alla trasformazione, viene elaborata in modo indipendente rispetto alle altre tuple. Possiamo allora indicare con tr_f la trasformazione corrispondente ad f , ma che è applicata ad una singola tupla.

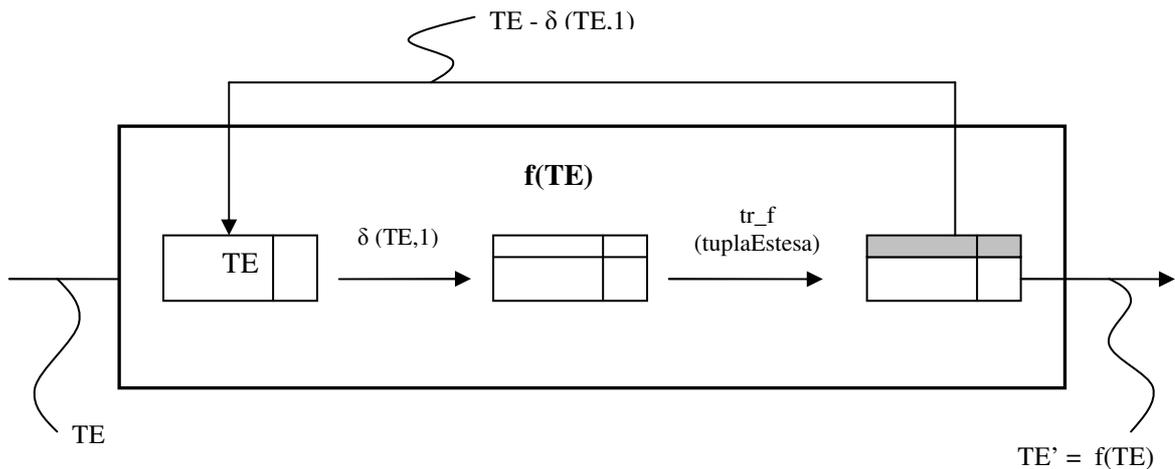


Figura 3.2: Schema logico della implementazione di una trasformazione “locale”

Come è messo in evidenza dallo schema, una trasformazione locale si può implementare estraendo la prima tupla della tabella in ingresso, alla quale viene applicata la trasformazione tr_f . La tupla, eventualmente modificata, viene poi reinserita nella tabella privata del primo elemento, dopo che essa è stata trasformata attraverso f . È evidente la natura ricorsiva di tale implementazione. Allora, nel definire la semantica operativa del linguaggio e delle trasformazioni, supporremo di avere a disposizione una macchina astratta provvista di ricorsione, e sfrutteremo questo meccanismo sviluppando le dimostrazioni per casi e per induzione.

L'implementazione di una trasformazione locale ha allora la forma

$$f(TE) = \text{Aggiungi}(tr_f(\text{prima_tupla_di_TE}), f(TE-1))$$

e

$$f(\text{tabella_vuota}) = \text{tabella_vuota}$$

dove $TE-1$ indica la tabella TE a cui è stata sottratta la prima tupla e la funzione *Aggiungi* inserisce la tupla trasformata, corrispondente al primo parametro, nella tabella corrispondente al secondo parametro. Il costrutto di riscrittura *ruleS* è un esempio di una tale trasformazione locale.

Il prodotto e la somma di trasformazioni locali

Su tutte le trasformazioni si può definire un'operazione di prodotto, data dalla composizione di funzioni:

$$f \circ g (TE) = g (f (TE))$$

Per le trasformazioni locali ha senso introdurre anche un'altra operazione. Si definisce la somma di trasformazioni nel seguente modo:

$$f + g (TE) = \text{Aggiungi} (\text{tr}_g (\text{tr}_f(\text{prima_tupla_di_TE})), \quad f+g(TE-1))$$

e

$$f + g (\text{tabella_vuota}) = \text{tabella_vuota}.$$

La somma di due trasformazioni indica quindi che non si scorre tutta la tabella prima di applicare la seconda trasformazione, ma si procede per singole tuple, applicando prima la trasformazione f e al risultato viene applicata immediatamente la trasformazione g .

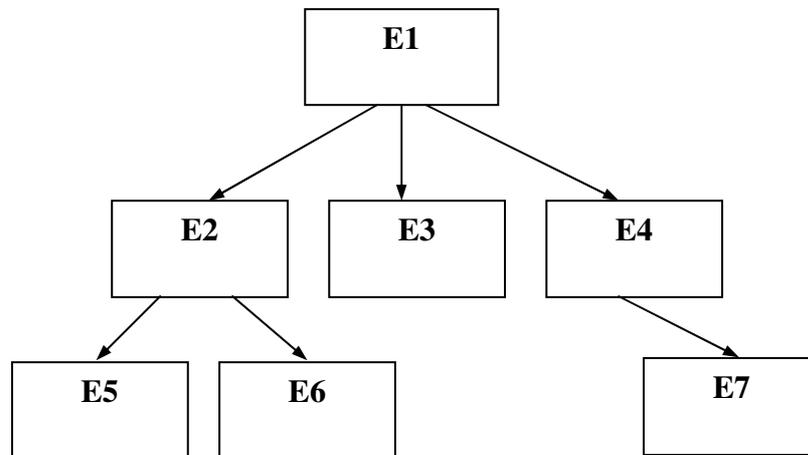
3.3.3 Semantica delle espressioni regolari

Di seguito viene ripresa la grammatica che definisce la sintassi delle espressioni regolari, sulla cui struttura induttiva si baserà la definizione semantica.

<ER>	::= <unione> <ER-semplice>
<unione>	::= <ER> " " <ER-semplice>
<ER-semplice>	::= <concatenazione> <ER-base>
<concatenazione>	::= <ER-semplice> <ER-base>
<ER-base>	::= <asterisco> <più> <parentesi-graffe> <ER-elementare>
<asterisco>	::= <ER-elementare> "*"
<più>	::= <ER-elementare> "+"
<parentesi-graffe>	::= <ER-elementare> "{" intero positivo "}" <ER-elementare> "{" intero positivo ", " intero positivo "}"
<ER-elementare>	::= <gruppo> <qualunque> <fine-stringa> <carattere> <insieme>
<gruppo>	::= "(" <ER> ")"
<qualunque>	::= "."
<fine-stringa>	::= "\$"
<carattere>	::= qualunque non metacarattere "\" metacarattere
<insieme>	::= <insieme-positivo> <insieme-negativo>
<insieme-positivo>	::= "[" <elementi-di-insieme> "]"
<insieme-negativo>	::= "[^" <elementi-di-insieme> "]"
<elementi-di-insieme>	::= <elemento-di-insieme> <elemento-di-insieme> <elementi- di-insieme>
<elemento-di-insieme>	::= <intervallo> <carattere>
<intervallo>	::= <carattere> "-" <carattere>

3.3.3.1 Semantica operativa delle espressioni regolari

Innanzitutto stabiliamo che un albero sintattico, in questo caso corrispondente ad una espressione regolare, possa essere rappresentato come una sequenza; sia ad esempio dato il seguente albero ER con i nodi etichettati:



L'albero grafico dell'esempio è rappresentato in termini di sequenze come:

$$ER == (E1, ((E2, ((E5, \text{null}), (E6, \text{null}))), (E3, \text{null}), (E4, (E7, \text{null}))))$$

Tutte le funzioni semantiche che seguono sono definite, come anticipato in precedenza, in accordo con il modello di astrazione delle *sequenze*.

La funzione semantica principale, che effettua il *pattern matching*, è **ERmatching(ER, S)**. Essa è applicata ad una espressione regolare *ER* e ad una stringa di ingresso *S*, e ne valuta la conformità nello schema. **ERmatching(ER, S)** è una funzione ricorsiva, dove *ER* indica l'albero sintattico associato alla stringa che rappresenta l'espressione regolare e *S* indica la stringa di input su cui si effettua il *pattern matching*. *S* è trattata come una sequenza di caratteri. La funzione restituisce una sequenza (s1, s2, s3), dove

- **s1** indica un valore booleano, che è positivo se e solo se esiste una sottostringa iniziale di *S*, ossia fino ad **s2**, che verifica il *pattern matching* con l'espressione regolare *ER*;

- **s2** è a sua volta una sequenza di caratteri, che rappresenta la parte residua della stringa di input S , la quale non ha ancora subito il confronto di *pattern matching*;
- **s3** memorizza, in ordine, la sequenza di sottostringhe di S che sono state associate ai raggruppamenti.

ERmatching(ER, S) is

case (Figli(ER)) of

{

(<ER>), (<ER-semplce>), (<unione>), (<concatenazione>), (<ER-base>),
 (<asterisco>), (<più>), (<parentesi-graffe>), (<ER-elementare>),
 (<gruppo>), (<qualunque>), (<fine-stringa>), (<carattere>), (<insieme-positivo>), (<insieme-negativo>), (<elemento-di-insieme>), (<intervallo>):

ERmatching(Figlio(ER,1), S)

(<ER1> "|" <ER-semplce>):

ERscegli(ERmatching(Figlio(ER,1), S), ERmatching(Figlio(ER,3), S))

(<ER-semplce> <ER-base>):

ERconcatenazione(Figlio(ER,1), Figlio(ER,2), S)

(<ER-elementare> "*"):

ERscegli(ERconcatenazione(Figlio(ER,1), ER, S), (true, S, null))

(<ER-elementare> "+"):

ERconcatenazione(Figlio(ER,1), Figlio(ER,1) "*", S)

(<ER-elementare> "{" n "}"):

```

if (n == 0) then (true, S, null)
      else ERconcatenazione(Figlio(ER,1), Figlio(ER,1) "{" n-1 "}", S )

```

```

(<ER-elementare> "{" n ", }" ) :
  ERconcatenazione(Figlio(ER,1) "{" n "}", Figlio(ER,1) "*", S)

```

```

(<ER-elementare> "{" n ", " m "}" ) :
  if (n == 0) then ERiterazioneLimitata(Figlio(ER,1), m, S)
      else ERconcatenazione(Figlio(ER,1), Figlio(ER,1) "{" n-1 ", " m-1
" }" ,S)

```

```

("(" <ER> ")" ) :
  ERcombinaConcat( (null,null,(S\δ(ERmatching(Figlio(ER,2), S),2))),
                    ERmatching(Figlio(ER,2), S) )

```

```

("." ) :
  if ((δ(S,1) == c) & (c ≠ ritorno_carrello))
      then (true, λ (S,1), null)
      else (false, null, null)

```

```

("$" ) :
  if (S == null )
      then (true, null, null)
      else (false, null, null)

```

```

(c): // * c può indicare un qualunque carattere o un metacarattere preceduto da "\"
  if (δ(S,1) == c )
      then (true, λ (S,1), null)
      else (false, null, null)

```

```

("[ <elementi-di-insieme> "]" ) :
  if ((δ(S,1) == c) & (ERappartiene(c, Figlio(ER, 2))))
      then (true, λ (S,1), null)

```

else (false, null, null)

("[" <elementi-di-insieme> "]") :

if (($\delta(S,1) == c$) & (**not** ERappartiene(c, Figlio(ER, 2))))

then (true, $\lambda(S,1)$, null)

else (false, null, null)

}

dove

Figli (ER) is

$\rho(\delta(ER,2),i), (1)$

Figlio(ER, i) is

$\delta(\delta(ER,2),i)$

ERscegli(S1, S2) is

if ($\delta(S1,1)$) **then** S1

else S2

ERconcatenazione(ER1, ER2, S) is

ERcombinaConcat(ERmatching(ER1, S) ,

ERmatching(ER2, $\delta(ERmatching(ER1, S),2)$)

ERcombinaConcat restituisce la sequenza S2, inserendo in testa al terzo elemento, il quale deve costituire a sua volta una sequenza, la sottosequenza rappresentata dal terzo elemento di S1

ERcombinaConcat(S1, S2) is

$\alpha(\lambda(S2,3)$,

$\alpha(\delta(S1,3), \delta(S2,3))$)

ERiterazioneLimitata restituisce la stessa terna di output della funzione *ERmatching*, dove il pattern rappresentato dall'espressione regolare *ER* può ricorrere al più *N* volte nella stringa *S*

ERiterazioneLimitata(ER, N, S) is

```

if (N == 0)
then if ( $\delta$ (ERmatching(ER,S),1) then (false, null, null)
else (true, S, null)
else if ( not  $\delta$ (ERmatching(ER,S),1)
then (true, S, null)
else ERcombinaConcat( ERmatching(ER, S) ,
ERiterazioneLimitata( ER, N-1,  $\delta$ 
(ERmatching(ER, S),2)) )

```

ERappartiene restituisce un valore booleano positivo se il carattere *C* appartiene alla classe di caratteri rappresentata dall'espressione regolare *ER*, e un valore negativo altrimenti

ERappartiene(C, ER) is

case (Figli(ER)) **of**

{

(<elemento-di-insieme>), (<intervallo>), (<carattere>):

ERappartiene(C, Figlio(ER,1))

(<elemento-di-insieme> <elementi-di-insieme>):

ERappartiene(C, Figlio(ER,1)) **or** ERappartiene(C, Figlio(ER,2))

(<carattere> "-" <carattere>):

(C >= Figlio(ER,1)) **&** (C <= Figlio(ER,3))

(c): // * *c* può indicare un qualunque carattere o un metacarattere preceduto da "\"

C == c

3.4 Implementazione

La semantica del linguaggio che daremo di seguito è una semantica operativa. Essa è definita utilizzando il metalinguaggio introdotto in [Tesi2], le cui operazioni primitive sono gli operatori su sequenze. Infatti le espressioni del linguaggio DPL sono modellate in termini di sequenze, così come lo sono state le espressioni del linguaggio MQL. Essendo comunque uniformi le rappresentazioni di oggetti diversi, risulta così agevole definire l'integrazione dei due linguaggi, sotto forma di estensione di MQL.

Il metalinguaggio di specifica è inoltre dotato di astrazione funzionale, di ricorsione, e possiede il costrutto *if then else* per il controllo di flusso. A questo punto è possibile considerare ogni costrutto di trasformazione del linguaggio come un operatore derivato dagli operatori elementari sulle sequenze. Per le definizioni di tali operatori elementari si rimanda all'appendice A e per una descrizione più dettagliata si rimanda a [Tesi2].

3.4.1 Le regole di riscrittura

Il meccanismo di riscrittura di stringhe, il quale rappresenta una delle caratteristiche fondamentali del nostro linguaggio, è offerto dal costrutto di riscrittura *RWRULES*. Tramite esso è possibile ricercare, marcare e sostituire delle stringhe, attraverso un meccanismo espressivo molto potente: il *pattern matching*.

Il costrutto *RWRULES* permette di specificare tre diversi tipi di regole di riscrittura, che saranno oggetto dei prossimi paragrafi.

Prima di entrare nei dettagli delle definizioni formali, ricordiamo la sintassi del costrutto di riscrittura.

Sintassi del costrutto di riscrittura

```
<Riscrittura> ::= APPLY RWRULES IN <tabella_estesa> TO <attributo>
                BEGIN
                    <Regole_riscrittura>
                END
```

$\langle \text{Regole_risrittura} \rangle ::= \langle \text{Regola} \rangle ; | \langle \text{Regola} \rangle ; \langle \text{Regole_risrittura} \rangle$
 $\langle \text{Regola} \rangle ::= \langle \text{RegolaS} \rangle | \langle \text{RegolaM} \rangle | \langle \text{RegolaT} \rangle$
 $\langle \text{RegolaS} \rangle ::= \langle \text{Pattern} \rangle \text{ INTO } \langle \text{Comando} \rangle [\text{ EXCEPTION } [\langle n \rangle]]$
 $\langle \text{Pattern} \rangle ::= \langle \langle \text{ER} \rangle^1 \rangle$
 $\langle \text{Comando} \rangle ::= \langle \text{Sostituzione} \rangle$
 | IF ($\langle \text{Condizione_rw} \rangle$) THEN $\langle \text{Comando} \rangle$
 | IF ($\langle \text{Condizione_rw} \rangle$) THEN $\langle \text{Comando} \rangle$ ELSE $\langle \text{Comando} \rangle$
 $\langle \text{Sostituzione} \rangle ::= \text{'stringa di sostituzione contenente variabili di raggruppamento } \$i \text{ e caratteri'}$
 $\langle \text{Condizione_rw} \rangle ::= \langle \text{Operando_rw} \rangle \langle \text{Operatore_confronto} \rangle \langle \text{Operando_rw} \rangle$
 | ($\langle \text{Condizione_rw} \rangle$) AND ($\langle \text{Condizione_rw} \rangle$)
 | ($\langle \text{Condizione_rw} \rangle$) OR ($\langle \text{Condizione_rw} \rangle$)
 | NOT $\langle \text{Condizione_rw} \rangle$
 $\langle \text{Operando_rw} \rangle ::= \text{'una variabile di raggruppamento } \$i \text{ oppure una sequenza di caratteri'}$
 $\langle \text{Operatore_confronto} \rangle ::= \text{una qualunque operazione di confronto } \theta \text{ tra stringhe oppure numeri}$
 $\langle \text{RegolaM} \rangle ::= \langle \text{Pattern} \rangle \text{ EXCEPTION } [\langle n \rangle]$
 $\langle \text{RegolaT} \rangle ::= \text{EVERY } \langle \text{Pattern} \rangle \text{ INTO } \langle \text{Sostituzione} \rangle [\text{ EXCEPTION } [\langle n \rangle]]$

3.4.2 La regola di riscrittura *ruleS*

La regola di riscrittura principale, *ruleS*, permette di sostituire tutte le stringhe che rappresentano il valore di un attributo di una tabella estesa. A questo scopo si utilizzano le espressioni regolari, introdotte nel paragrafo 3.3.3, che servono per esprimere un *pattern*, ossia uno schema di costruzione di una stringa generica. L'applicazione della regola di riscrittura consiste nell'effettuare un *pattern matching* tra l'espressione regolare della regola, e le stringhe rappresentanti l'attributo in questione. Se esso dovesse fallire per qualche tupla, la sostituzione non avviene, e nel caso in cui fosse

¹ ER rappresenta un'espressione regolare, come quelle utilizzate nel linguaggio Perl. Le espressioni regolari valide nel formalismo DPL sono definite nei paragrafi 2.4 e 3.3.3.

specificato il tag *EXCEPTION*, la tupla viene marcata, ossia estesa con le informazioni sull'avvenuta eccezione.

La stringa di sostituzione può essere, nel caso più semplice, una stringa costante, ma la vera potenza di questo costrutto consiste nella possibilità di definire una nuova stringa a partire da quella data in ingresso. Infatti il meccanismo di pattern matching non solo verifica che la stringa in ingresso soddisfi un certo schema di costruzione, ma allo stesso tempo lega, quando richiesto, delle sottostringhe a cosiddette *variabili di raggruppamento*. Allora la nuova stringa può essere composta a piacere, a partire da componenti della stringa in ingresso. È inoltre possibile esprimere una condizione, attraverso l'espressione *if then else*, in modo che la sostituzione viene effettuata solamente in alcune tuple.

Come abbiamo già anticipato, la semantica operativa è definita in termini degli operatori algebrici su sequenze introdotti in [Tesi2]. Ogni costrutto di trasformazione è a questo punto definito come una *operazione derivata* sulle sequenze. Il costrutto di riscrittura *ruleS* viene nel formalismo DPL applicato ad una tabella estesa, la quale verrà poi modificata opportunamente. Allora l'operazione derivata associata al costrutto *ruleS* verrà concordemente applicata ad un oggetto di tipo TabEstesa e darà come risultato di nuovo un oggetto di tipo TabEstesa, le cui sottocomponenti, ancora sequenze, sono trasformate in base alla regola di riscrittura.

Se fosse richiesta la marcatura delle eccezioni, l'estensione della tabella risultato conterrà oltre alle metainformazioni presenti nella tabella in ingresso, anche le informazioni su eventuali eccezioni verificatesi durante questa ultima trasformazione per riscrittura.

Sintassi concreta

```
APPLY RWRULES IN tabella TO attributo
BEGIN
    ...
    << pattern >> INTO comando [EXCEPTION [marca]];
    ...
END
```

Sintassi astratta

RuleS(tabella, attributo, pattern, comando[, marca])

Implementazione

RuleS(tabellaEstesa, attributo, pattern, comando)

≡ F_rwS(tabellaEstesa, null, attributo, pattern, comando)

RuleS(tabellaEstesa, attributo, pattern, comando, marca)

≡ F_rwS(tabellaEstesa, marca, attributo, pattern, comando)

dove

F_rwS è la funzione ricorsiva che implementa il costrutto ‘ruleS’ ; si noti che essa rispetta la struttura di definizione delle trasformazioni locali.

F_rwS(T, M, A, P, C) is

if ($\delta(\epsilon(\text{TabEstesa}, \text{Valori}, T), 1) == \text{null}$)

then T

else Aggiungi(Tr_rwS($\delta(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, T), 1), 1)$,
 $\tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, T), 1), A)$,
 $\delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, T), 1)$,
P, C, M),

F_rwS(RiduciTabella(T), M, A, P, C))

Tr_rwS effettua la vera e propria trasformazione della tupla TP, applicata all’attributo in posizione POS. NA indica il numero di attributi della tabella; P, C ed E sono i parametri della trasformazione effettuata dalla regola di riscrittura ‘ruleS’.

```

Tr_rwS(TP, POS, NA, P, C, M) is
if (M == null)
  then NuovaTupla(TP, NuovoValoreS( $\delta$ (TP,POS), P, C), POS , null , NA)
else if (M == "NO_MARK")
  then NuovaTupla(TP, NuovoValoreS( $\delta$ (TP,POS), P, C), POS ,
    if (EccezioneS( $\delta$ (TP,POS), P))
      then "rwS"+P+C+  $\delta$ (TP,POS)
      else null ,
    NA)
else NuovaTupla(TP, NuovoValoreS( $\delta$ (TP,POS), P, C), POS ,
  if (EccezioneS( $\delta$ (TP,POS), P))
    then "rwS"+P+C+  $\delta$ (TP,POS)+ "Codice: "+M
    else null ,
  NA)

```

NuovoValore restituisce la stringa che dovrà sostituire *S*, in base all'espressione regolare rappresentata da *P* e in base al comando rappresentato da *C*

```

NuovoValoreS(S, P, C) is
if ( $\delta$  (ERmatching(AlberoSintattico(P), S),1))
  then ApplicaComando( $\delta$ (ERmatching(AlberoSintattico(P), S),3), C, S)
  else S

```

EccezioneS(S,P) is

```

not  $\delta$  (ERmatching(AlberoSintattico(P), S),1)

```

ApplicaComando controlla le eventuali condizioni sulle sottostringhe della stringa di ingresso *S*, individuate attraverso raggruppamenti contenuti in *G*, e fornisce la nuova stringa di sostituzione in base all'espressione contenuta nel comando *C*

ApplicaComando(G, C, S) is

case (C) of

{

(if *Condizione* then *C1* else *C2*):

if (ValutaCondizione_rw(Figlio(AlberoSintattico(C), 2), G))

then ApplicaComando(G, C1, S)

else ApplicaComando(G, C2, S)

(if *Condizione* then *C1*):

if (ValutaCondizione_rw(Figlio(AlberoSintattico(C), 2), G))

then ApplicaComando(G, C1, S)

else S

(*Sost*): */* Sost indica una stringa formata da una sequenza di caratteri e di variabili di raggruppamento (\$n)*

AggiornaStringa(*Sost*, G)

}

ValutaCondizione_rw restituisce il valore booleano corrispondente alla valutazione della condizione *Cond*, rappresentata attraverso il suo albero sintattico

ValutaCondizione_rw (Cond, G) is

case (Figli(Cond)) of

{

(<Operando_rw> <Operatore_confronto > <Operando_rw>) :

Semantica(Sem_Operando(Figlio(Cond, 1), G)

Semantica(Figlio(Cond, 2))

Sem_Operando(Figlio(Cond, 3), G))

((“ <Condizione_rw > ”) **AND** (“ <Condizione_rw > ”)) :

ValutaCondizione_rw (Figlio(Cond, 2), G) &_{sem}

ValutaCondizione_rw (Figlio(Cond, 4), G)

(“(“ <Condizione_rw >”) **OR** (“ <Condizione_rw >”)) :

ValutaCondizione_rw (Figlio(Cond, 2), G) \parallel_{sem}

ValutaCondizione_rw (Figlio(Cond, 4), G)

(“**NOT** “ <Condizione_rw >) :

not_{sem} ValutaCondizione_rw (Figlio(Cond, 2), G)

}

Sem_Operando(OP,G) is

case (OP) **of**

{

(“(\$”n”)”: $\delta(G, n)$

(S) : S */* S indica una sequenza di caratteri e rappresenta un valore costante di stringa*

}

AggiornaStringa restituisce la stringa corrispondente alla stringa di sostituzione Sost, in cui ogni occorrenza di variabile di raggruppamento \$i è sostituita con il corrispondente valore, contenuto nella sequenza G

AggiornaStringa(Sost,G) is

if (Sost == null)

then null

else if ((δ (Sost, 1) == “(“ & (δ (Sost, 2) == “\$“ & (δ (Sost, 4) == “)“))

then α (δ (G, δ (Sost, 3)) , AggiornaStringa($\lambda(\lambda(\lambda(\lambda$ (Sost,1),1),1),1), G)

else α (δ (Sost, 1) , AggiornaStringa(λ (Sost,1), G)

3.4.3 La regola di riscrittura *ruleM*

Una seconda possibile clausola all'interno del corpo del costrutto *APPLY RWRULES* è costituita dalla regola *ruleM*. Essa non effettua alcuna modifica sui valori dell'attributo in questione, ma serve solamente per verificare che sia rispettato un dato schema di costruzione delle stringhe, chiamato *pattern*. Ogni qual volta il *pattern matching* dovesse fallire, la tupla viene “marcata” e l'eccezione sollevata viene memorizzata nell'estensione della tabella. I dati che tengono traccia dell'eccezione sollevata sono: il tipo di costrutto (in questo caso *ruleM*), i suoi parametri attuali (la tabella, l'attributo, il *pattern*), il valore corrente dell'attributo, e se fosse specificato, il codice di marcatura contenuto in *marca*.

Sintassi concreta

```

APPLY RWRULES IN tabella TO attributo
BEGIN
    ...
    << pattern >> EXCEPTION [marca];
    ...
END

```

Sintassi astratta

RuleM(*tabella*, *attributo*, *pattern*, *marca*)

Implementazione

RuleM(*tabellaEstesa*, *attributo*, *pattern*, *marca*)
 \equiv *F_rwM*(*tabellaEstesa*, *marca*, *attributo*, *pattern*)

EccezioneM restituisce l'esito del pattern matching tra la stringa in ingresso *S* ed il pattern *P*

EccezioneM(S,P) is

not δ (ERmatching(AlberoSintattico(P), S),1)

3.4.4 La regola di riscrittura *ruleT*

All'interno del corpo del costrutto *APPLY RWRULES* può apparire una terza possibile clausola di riscrittura, che chiameremo *ruleT*. Così come nella prima clausola vista, è presente un'espressione regolare che descrive uno schema di costruzione di una stringa, ossia un pattern. Tuttavia in questo caso il pattern non si riferisce all'intero contenuto dell'attributo. Il pattern matching viene effettuato ripetutamente e la stringa di sostituzione andrà a rimpiazzare tutte le occorrenze del pattern incontrate nella stringa rappresentante il valore attuale dell'attributo. Anche nella clausola *ruleT* la stringa di sostituzione è costituita da caratteri e da variabili di raggruppamento $\$n$ racchiuse da parentesi tonde, come ad esempio “abc(\$1)%&\$(2)123(\$3)!”. Se si volesse che la sequenza di caratteri ($\$1$) non fosse trattata come *meta-stringa* rappresentante il valore del primo raggruppamento, ma come una normale parola di sostituzione, sarà sufficiente far precedere il carattere ‘\’.

Se è specificato il tag *EXCEPTION* e nessuna occorrenza del pattern dovesse essere trovata all'interno di un attributo, allora nell'estensione della tabella la tupla verrà “estesa” con le informazioni di marcatura/eccezione: il tipo di costrutto che ha sollevato eccezione, i suoi parametri attuali, il valore corrente dell'attributo, e se fosse specificato, il codice di marcatura contenuto in *marca*.

Sintassi concreta

APPLY RWRULES IN *tabella* TO *attributo*

BEGIN

...

```

    EVERY << pattern >> INTO << sostituzione >> [EXCEPTION [marca]];
    ...
END

```

Sintassi astratta

RuleT(tabella, attributo, pattern, sostituzione[, marca])

Implementazione

RuleT(tabellaEstesa, attributo, pattern, sostituzione)

\equiv F_rwT(tabellaEstesa, null, attributo, pattern, sostituzione)

RuleT(tabellaEstesa, attributo, pattern, sostituzione, marca)

\equiv F_rwT(tabellaEstesa, marca, attributo, pattern, sostituzione)

dove

F_rwT è la funzione ricorsiva che implementa la clausola ‘ruleT’ ; si noti che essa rispetta la struttura di definizione delle trasformazioni locali.

F_rwT(T, M, A, P, S) is

if ($\delta(\epsilon(\text{TabEstesa}, \text{Valori}, T), 1) == \text{null}$)

then T

else Aggiungi(Tr_rwT($\delta(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, T), 1), 1)$,

$\tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, T), 1), A)$,

$\delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, T), 1)$,

P, S, M),

F_rwT(RiduciTabella(T), M, A, P, S))

Tr_{rwT} effettua la vera e propria trasformazione della tupla *TP*, applicata all'attributo in posizione *POS*. *NA* indica il numero di attributi della tabella; *P*, *S* ed *M* sono i parametri della trasformazione effettuata dalla regola di riscrittura 'ruleT'.

Tr_{rwT}(TP, POS, NA, P, S, M) is

if (M == null)

then NuovaTupla(TP, NuovoValoreT(δ (TP,POS), P, S), POS, null, NA)

else if (M == "NO_MARK")

then NuovaTupla(TP, NuovoValoreT(δ (TP,POS), P, S), POS,

if (EccezioneT(δ (TP,POS), P, S))

then "rwT"+P+S+ δ (TP,POS)

else null,

NA)

else NuovaTupla(TP, NuovoValoreT(δ (TP,POS), P, S), POS,

if (EccezioneT(δ (TP,POS), P, S))

then "rwT"+P+S+ δ (TP,POS)+"Codice: "+M

else null,

NA)

NuovoValoreT restituisce la stringa che dovrà sostituire *S*, in base all'espressione regolare rappresentata da *P* e alla stringa di sostituzione rappresentata da *SOST*

NuovoValoreT(S, P, SOST) is

δ (TraduciOccorrenzeDi(AlberoSintattico(P), SOST, S, false),2)

EccezioneT(S,P,SOST) is

not δ (TraduciOccorrenzeDi(AlberoSintattico(P), SOST, S, false),1)

TraduciOccorrenzeDi è una funzione ricorsiva che sostituisce all'inizio della stringa di ingresso *S* l'eventuale occorrenza del pattern, rappresentato dall'albero sintattico di

una espressione regolare ER , con la stringa di sostituzione $SOST$. Essa restituisce una coppia, dove il primo è un valore booleano che è vero se il pattern matching con ER ha avuto successo almeno una volta all'interno della stringa in ingresso S , ed è falso altrimenti. Il secondo valore di uscita contiene la stringa in ingresso S in cui tutte le occorrenze del pattern sono state rimpiazzate con la stringa di sostituzione $SOST$. $MATCH$ serve per tenere traccia dell'esito dei pattern matching ai passi precedenti.

TraduciOccorrenzeDi(ER,SOST,S,MATCH) is

```

if (S==null)
  then (MATCH, null)
  else if ( $\delta$  (Ermatching(ER,S),1))
    then ( true,
           $\alpha$ ( AggiornaStringa(SOST,  $\delta$  (Ermatching(ER,S),3) ),
               $\delta$ (TraduciOccorrenzeDi(ER, SOST,  $\delta$ 
                (Ermatching(ER,S),2), true),2)))
    else ( MATCH,
           $\alpha$ (  $\delta$  (S,1) ),
           $\delta$ (TraduciOccorrenzeDi(ER, SOST,  $\lambda$  (S,1) , MATCH),2)))

```

AggiornaStringa restituisce una stringa, a partire dalla stringa di sostituzione $SOST$, dove le variabili di raggruppamento in $SOST$ sono sostituite con i valori opportuni contenuti nella sequenza G . G contiene nel giusto ordine i valori delle variabili di raggruppamento.

AggiornaStringa(SOST,G) is

```

if (SOST == null)
  then null
  else if (( $\delta$ (SOST, 1) == "(" &  $\delta$ (SOST, 2) == "$" &  $\delta$ (SOST, 4) == ")")
    then  $\alpha$ (  $\delta$ (G,  $\delta$ (SOST, 3) ), AggiornaStringa( $\lambda$ ( $\lambda$ ( $\lambda$ (SOST,1),1),1),1),G))
    else  $\alpha$ (  $\delta$ (SOST, 1) ), AggiornaStringa(  $\lambda$ (SOST,1), G )

```

3.4.5 Le trasformazioni di schema

Le trasformazioni di schema effettuano una modifica soprattutto a livello di schema relazionale di una tabella, e solo secondariamente adeguano il contenuto “estensionale” della tabella al nuovo schema. Quando possibile, si è cercato, nell’implementazione di tali costrutti, di separare queste due fasi, anticipando le modifiche di schema. Ciò ha permesso di estendere le proprietà delle trasformazioni locali anche ad alcune trasformazioni di schema, chiamate perciò *trasformazioni locali di schema*. Escluso il costrutto FOLD, appartengono a questa classe tutti i costrutti per le trasformazioni di schema:

- DIVIDE
- SPLIT
- MERGE
- NEWATTRIBUTE
- REMOVE
- RENAME .

Le trasformazioni di schema sono previste in tutti i linguaggi per basi di dati, ma è opportuno evidenziare la potenzialità espressiva dei costrutti del linguaggio DPL: ogni condizione booleana può anche essere rappresentata da un confronto per “similitudine”, e non solo da una “uguaglianza pura”. Si tenga presente che in un ambiente per estrazione di conoscenza, sono presenti tutte le problematiche di eterogeneità dei dati, dovute al fatto che essi provengono da diverse sorgenti. Inoltre, poter esprimere una condizione di similitudine, rappresenta un ulteriore strumento per affrontare il problema dei dati “sporchi”.

Immaginiamo il semplice caso in cui si voglia che la condizione booleana indichi che l’attributo *Indirizzo* sia uguale a “Via G. Garibaldi”. Una elementare condizione di uguaglianza tra stringhe potrebbe non catturare diversi casi in cui, in realtà, la condizione vale, come ad esempio “via G. Garibaldi”, “Via Giuseppe Garibaldi”, “Via G. Garibaldi, n. 15”. Invece con il meccanismo di pattern matching su stringhe, di cui il nostro linguaggio è provvisto, possiamo anche verificare una eventuale “similitudine”

tra due stringhe, senza dover necessariamente conoscerle anticipatamente in modo esatto. Le espressioni regolari permettono infatti di rappresentare una stringa attraverso il suo schema di costruzione, piuttosto che attraverso una sua riproduzione esatta.

Implementazione delle trasformazioni locali di schema

Per poter estendere anche alle trasformazioni locali di schema qualche utile proprietà, è opportuno seguire nella loro implementazione il seguente modello:

$$\begin{aligned} \mathbf{h(TE)} &= \mathbf{Funzione_locale(Modifica_di_schema(TE, parametri), parametri)} \\ &= h_s \circ h_l(TE) \end{aligned}$$

dove *Modifica_di_schema* (indicata anche con h_s) opera soltanto sugli elementi *Attributi* e *NumAttributi* di un oggetto di tipo *TabEstesa*. *F_locale* (indicata anche con h_l) è una trasformazione locale tradizionale, la quale opera sull'elemento *Valori*, procedendo tupla per tupla.

Anche su questa classe di trasformazioni è definita l'operazione di somma, indotta (quindi compatibile) dalla somma su trasformazioni locali. Essa è definita nel seguente modo:

$$h + f(TE) = (h_s \circ h_l) + f(TE) = h_s \circ (h_l + f)(TE) .$$

Si noti che la somma è definita solamente quando il secondo termine è una trasformazione locale elementare, mentre non è definita quando questo secondo termine comporta una modifica a livello di schema relazionale.

Le trasformazioni locali di schema godono di alcune proprietà interessanti, che saranno enunciate e dimostrate nel prossimo capitolo. Nei paragrafi 3.4.6 fino a 3.4.12 saranno trattati più in dettaglio i singoli costrutti per le trasformazioni di schema.

3.4.6 DIVIDE

Il costrutto DIVIDE effettua soprattutto una modifica a livello di schema relazionale, la quale comporta naturalmente un'opportuna trasformazione nel contenuto "estensionale" della tabella. Infatti si dice che il costrutto DIVIDE appartiene alla classe delle *trasformazioni di schema*.

In particolare, il costrutto "di schema" DIVIDE comprende una condizione, in base alla quale il contenuto di un attributo viene distribuito tra due colonne. Vediamo subito questo costrutto più in dettaglio.

Sintassi :

```

<Divisione> ::= DIVIDE <attributo> IN <tabella_estesa>
                IF ( <Condizione_div> )
                PUT INTO <stringa>2

<Condizione_div> ::=      <Operatore_confronto>      <Operando_div>      ,
<Operando_div>
                | MATCH <ER3>
                | NOT <Condizione_div>
                | (<Condizione_div>) (AND|OR) (<Condizione_div>)

<Operando_div> ::= SOURCE | <stringa> | <numero>

<Operatore_confronto> ::= una qualunque operazione di confronto θ tra
                        stringhe oppure numeri

```

² Non è previsto il tag *EXCEPTION* nel costrutto DIVIDE.

³ ER rappresenta un'espressione regolare, simile a quelle utilizzate nel linguaggio Perl.

Descrizione del costrutto :

Il tag *INTO* <stringa> serve per definire il nome dell'attributo con cui lo schema della tabella viene esteso. Il tag *IF* <Condizione_div> indica la condizione che viene verificata, per ogni tupla, sul valore dell'attributo specificato in *DIVIDE* <attributo>. In base all'esito della guardia condizionale, il contenuto di quest'ultimo attributo viene distribuito: se la condizione vale, allora il valore passa dalla colonna origine alla nuova colonna, relativa all'attributo specificato attraverso il tag *INTO* <stringa>; altrimenti il valore rimane nella posizione originaria. Il risultato dell'operazione è una tabella estesa modificata nello schema: essa possiede un nuovo attributo, e la relativa parte di estensione, necessaria per il meccanismo di marcatura. Per quanto riguarda il contenuto di ogni tupla, il valore dell'attributo *origine*, ossia quello su cui è definita la trasformazione, sarà contenuto nell'attributo *origine*, oppure nel nuovo attributo della tabella.

Il tag *SOURCE* viene utilizzato all'interno della condizione, e rappresenta una variabile locale al costrutto, che indica, per ogni tupla, il contenuto corrente dell'attributo *origine*.

Sintassi astratta :

DIVIDE *Attributo* *IN* *Tabella*

IF *Condizione*

PUT INTO *NuovoAttributo*

≡

DIVIDE(*Tabella*, *Attributo*, *NuovoAttributo*, *Condizione*)

Implementazione :

DIVIDE(Tab, Att, NAtt, Cond) is

F_divide(Fschema_AggiungiColonna(Tab, NAtt), Att, Cond)

dove

F_divide effettua la trasformazione, tupla per tupla, per quanto riguarda il contenuto della tabella; lascia inalterato lo schema relazionale

F_divide(Tab, Att, Cond) is

if ($\delta(\epsilon(\text{TabEstesa}, \text{Valori}, \text{Tab}), 1) == \text{null}$)

then Tab

else Aggiungi(Tr_divide ($\delta(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, \text{Tab}), 1), 1)$,
 $\tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, \text{Tab}), 1), \text{Att})$,
 $\delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, \text{Tab}), 1)$
 Cond),

F_divide (RiduciTabella(Tab), Att, Cond))

Tr_divide effettua la vera e propria trasformazione della tupla TP, applicata all'attributo in posizione PosA.

Tr_divide (TP, PosA, PosNA, Cond) is

case (Cond) of

{

(θ SeqOperandi):

if (Sem(θ , SeqOperandi, $\delta(\text{TP}, \text{PosA})$))

then ApplicaDiv1(TP, PosA, PosNA)

else ApplicaDiv2(TP, PosA, PosNA)

(MATCH EsprReg):

```

if ( $\delta(\text{ERmatching}(\text{AlberoSintattico}(\text{EsprReg}), \delta(\text{TP}, \text{PosA})), 1)$ )
    then ApplicaDiv1(TP, PosA, PosNA)
    else ApplicaDiv2(TP, PosA, PosNA)

```

(θ SeqCondizioni): /* θ indica un'operatore logico AND, OR oppure NOT

```

if (Sem( $\theta$ , SeqCondizioni)) then ApplicaDiv1(TP, PosA, PosNA)
    else ApplicaDiv2(TP, PosA, PosNA)

```

}

*ApplicaDiv1 estende la tupla TP con un nuovo elemento in coda agli attributi (la cui posizione è rappresentata da PosNA), e con un elemento null in coda all'estensione della tupla (ossia in posizione $2 * \text{PosNA}$); inoltre sposta il contenuto della posizione posA in posizione PosNA*

ApplicaDiv1(TP, PosA, PosNA) is

```

 $\zeta$  (  $\delta(\text{TP}, \text{PosA})$ ,
     $\alpha$  ( $\varphi(\text{null}, \text{TP}, \text{PosA})$ , (null)),
    PosNA)

```

*ApplicaDiv2 estende la tupla TP con un nuovo elemento in coda agli attributi (la cui posizione è rappresentata da PosNA), e con un elemento null in coda all'estensione della tupla (ossia in posizione $2 * \text{PosNA}$)*

ApplicaDiv2(TP, PosA, PosNA) is

```

 $\zeta$  ( null,
     $\alpha$  (TP, (null)),
    PosNA)

```

□

3.4.7 SPLIT

Il costrutto SPLIT serve per spezzare una stringa e distribuire i due valori così ottenuti su due attributi diversi. Il punto di “spezzamento” viene identificato come la prima occorrenza di una sottostringa specificata dall’utente attraverso una espressione regolare, la quale apparterrà interamente alla seconda parte dello *split*.

Se consideriamo ad esempio la stringa “Via G. Garibaldi, n. 15”, e la spezziamo utilizzando il costrutto SPLIT con l’espressione regolare separatrice “n.”, allora il risultato dello spezzamento è formato dalle due stringhe “Via G. Garibaldi, ” e “n. 15”. Può essere in genere necessario ripulire le sottostringhe così ottenute con i costrutti di riscrittura, per eliminare segni di interpunzione e spazi vuoti. La combinazione di questi due costrutti è di particolare utilità quando è necessario uniformare dati con formato eterogeneo, perché provenienti da sorgenti diverse.

Sintassi :

```
<Split> ::= SPLIT <attributo> IN <tabella_estesa>
           AT ( <Espressione_regolare> )
           PUT INTO <stringa> AND <stringa>
           [EXCEPTION [<n>]]
```

Descrizione del costrutto :

Il tag *AT* (<Espressione_regolare>) rappresenta la sottostringa separatrice. Il valore dell’attributo di partenza è specificato in <attributo>, mentre le due ricorrenze di <stringa> nel tag *PUT INTO* servono per definire i due nuovi attributi con cui la tabella viene estesa; in seguito allo *splitting*, la prima parte del risultato viene inserita nel primo attributo, e la rimanente parte nel secondo attributo.

Il tag *EXCEPTION* permette di marcare le tuple in cui non è stato identificato il punto di spezzamento.

Sintassi astratta :

SPLIT *Attributo* IN *Tabella*

AT *EspressioneRegolare*

PUT INTO *NuovoAttributo1* AND *NuovoAttributo2*

[EXCEPTION [*Marca*]]

≡

SPLIT(*Tabella*, *Attributo*, *NuovoAttributo1*, *NuovoAttributo2*,
EspressioneRegolare, *Marca*)

dove

- *Marca* == “ null “ se non c’è il tag EXCEPTION;
- *Marca* == “ NO_MARK “ se c’è il tag EXCEPTION ma non è specificato un codice di marcatura;
- *Marca* == *Cod* se le eccezioni sono marcate con il codice *Cod*;

Implementazione :

SPLIT(*T*, *Att*, *NAtt1*, *NAtt2*, *ER*, *M*) is

F_split (

Fschema_AggiungiColonna(

Fschema_AggiungiColonna(*Tab*, *NAtt1*),

NAtt2),

Att, *ER*, *M*, *NAtt1*, *NAtt2*)

dove

F_split effettua la trasformazione, tupla per tupla, per quanto riguarda il contenuto della tabella; lascia inalterato lo schema relazionale

F_split(T, A, ER, M, NAtt1, NAtt2) is

if ($\delta(\epsilon(\text{TabEstesa}, \text{Valori}, T), 1) == \text{null}$)

then T

else Aggiungi(Tr_split ($\delta(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, T), 1), 1)$,
 $\tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, T), 1), A)$,
 $\delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, T), 1) - 1$,
 $\delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, T), 1)$,
 $\delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, T), 1)$,
ER, M, NAtt1, NAtt2) ,

F_split (RiduciTabella(T), A, ER, M, NAtt1, NAtt2))

Tr_split applica lo splittig alla tupla TP: spezza la stringa dell'elemento in posizione PosA,

in corrispondenza del primo pattern matching con ER

Tr_split (TP, PosA, PosNA1, PosNA2, NA, ER, M, NAtt1, NAtt2) is

if (M == null)

then NuovaTupla2(TP,

$\delta(\text{ApplicaTaglio}(\text{ER}, \text{null}, \delta(\text{TP}, \text{PosA})), 2)$, PosNA1,

$\delta(\text{ApplicaTaglio}(\text{ER}, \text{null}, \delta(\text{TP}, \text{PosA})), 3)$, PosNA2,

null , PosA, NA)

else if (M == "NO_MARK")

then NuovaTupla2(TP,

$\delta(\text{ApplicaTaglio}(\text{ER}, \text{null}, \delta(\text{TP}, \text{PosA})), 2)$, PosNA1,

$\delta(\text{ApplicaTaglio}(\text{ER}, \text{null}, \delta(\text{TP}, \text{PosA})), 3)$, PosNA2,

```

    if ( not  $\delta$ ( ApplicaTaglio( ER, null,  $\delta$ (TP,PosA)),1))
        then “split”+ER+Natt1+Natt2+  $\delta$ (TP,POS)
        else null ,
    PosA, NA)
else NuovaTupla2( TP,
     $\delta$ ( ApplicaTaglio( ER, null,  $\delta$ (TP,PosA)),2), PosNA1,
     $\delta$ ( ApplicaTaglio( ER, null,  $\delta$ (TP,PosA)),3), PosNA2,
    if ( not  $\delta$ ( ApplicaTaglio( ER,null,  $\delta$ (TP,PosA)),1))
        then ““split”+ER+Natt1+Natt2+  $\delta$ (TP,POS)
        +”Codice: ”+M
        else null ,
    PosA, NA)

```

NuovaTupla2 inserisce il valore NV1 nella tupla TP in posizione Pos1, e il valore NV2 in posizione Pos2. Inoltre inserisce il valore null nelle posizioni NV1+NA e NV2+NA. Nel caso in cui E non fosse null, lo aggiunge, concatenandolo al valore corrente, in posizione Pos0+NA

NuovaTupla2(TP, NV1, Pos1, NV2, Pos2, E, Pos0, NA) is

```

    if (E == null)
        then  $\zeta$  ( null,  $\zeta$  (null,  $\zeta$  (NV2,  $\zeta$ (NV1, TP, Pos1), Pos2), Pos1+NA), Pos2+NA)
    else  $\eta$  ( E ,
         $\zeta$  ( null,  $\zeta$  (null,  $\zeta$  (NV2,  $\zeta$ (NV1, TP, Pos1), Pos2), Pos1+NA),
            Pos2+NA),
        Pos0+NA)

```

ApplicaTaglio ricerca la prima occorrenza dell’espressione regolare ER nella stringa S1@S2 (all’inizio S1 è vuota), ed effettua il taglio (split) in quel punto. Restituisce una terna: il primo valore indica l’esito della ricerca, il secondo valore contiene la prima parte del taglio, e il terzo valore contiene la seconda parte del taglio. I caratteri

vengono spostati ricorsivamente da $S2$ a $S1$ fino a quando non si è raggiunta la fine di $S2$ oppure il pattern matching ha avuto successo

ApplicaTaglio(ER, S1, S2) is

```

if (S2==null)
  then (false, S1, S2)
  else if ( $\delta$  (Ermatching(ER,S2),1))
    then ( true, S1, S2)
    else ApplicaTaglio(ER,
       $\alpha$ ( S1, ( $\delta$  (S2,1)) ),
       $\lambda$  (S2,1))

```

3.4.8 MERGE

Questo costrutto aumenta lo schema relazionale di una tabella estesa con un nuovo attributo. Il valore che esso assume è formato dalla fusione, intesa come concatenazione di stringhe, di due attributi, i quali non vengono rimossi dalla tabella. Sarà eventualmente necessario rimuoverli successivamente in maniera esplicita. Si è preferita questa alternativa, per mantenere le informazioni di marcatura dei due attributi di fusione. Infatti la trasformazione della tabella attraverso il *MERGE* non assegna alcuna informazione di marcatura al nuovo attributo risultato.

Sintassi :

```

<Fusione> ::= MERGE <attributo> AND <attributo>
             IN <tabella_estesa>
             PUT INTO <stringa>

```

Descrizione del costrutto :

I due attributi di fusione sono specificati in *MERGE* <attributo> *AND* <attributo>, mentre il nome del nuovo attributo risultato è specificato nel tag *PUT INTO* <stringa>.

Non è previsto il tag *EXCEPTION*, e il nuovo attributo non ha alcuna informazione di marcatura associata.

Sintassi astratta :

MERGE *Attributo1* *AND* *Attributo2*

IN *Tabella*

PUT INTO *NuovoAttributo*

≡

MERGE(*Tabella*, *Attributo1*, *Attributo2*, *NuovoAttributo*)

Implementazione :

***MERGE*(*Tab*, *Att1*, *Att2*, *NAtt*) is**

F_merge(*Fschema_AggiungiColonna*(*Tab*, *NAtt*), *Att1*, *Att2*)

F_merge procede tupla per tupla e concatena il valore dell'attributo *A1* e il valore dell'attributo *A2*, inserendo il valore risultante nella ultima colonna, relativa al nuovo attributo con cui la tabella è stata precedentemente estesa.

***F_merge*(*T*, *A1*, *A2*) is**

if ($\delta(\epsilon(\text{TabEstesa}, \text{Valori}, T), 1) == \text{null}$)

then *T*

else *Aggiungi*(*Tr_merge*($\delta(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, T), 1), 1)$,

$\tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, T), 1)$, *A1*),

$$\begin{aligned} & \tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, T), 1), A2), \\ & \delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, T), 1)), \\ \text{F_merge} (& \text{RiduciTabella}(T), A1, A2)) \end{aligned}$$

*Tr_merge fonde i due attributi di fusione relativamente alla tupla TP, inserisce il risultato in posizione NA, e inserisce il valore null in posizione 2*NA*

Tr_merge(TP, Pos1, Pos2, NA) is

$$\begin{aligned} & \zeta(\text{null}, \\ & \zeta(\alpha(\delta(\text{TP}, \text{Pos1}), \delta(\text{TP}, \text{Pos2})), \text{TP}, \text{NA}) \\ & 2*\text{NA}) \end{aligned}$$

3.4.9 RENAME

I tre costrutti RENAME, REMOVE e NEWATTRIBUTE sono le tre trasformazioni di schema di base, indispensabili in ogni linguaggio per basi di dati. Essi operano su tabelle estese, e appartengono alla classe di *trasformazioni locali di schema*.

Riportiamo brevemente la sintassi e la semantica operativa dei tre costrutti.

Sintassi :

**<Rinominazione> ::= RENAME <attributo> IN <tabella_estesa>
AS <stringa>**

Implementazione :

RENAME(T, A, N) is

Fschema_RinominaColonna(T, A, N)

dove

Fschema_RinominaColonna(T, A, N) is

$\gamma(\text{TabEstesa}, ((\varphi(N,$

$\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, T), 1),$

$\tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, T), 1), A)),$

$\delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, T), 1)),$

$\delta(\epsilon(\text{TabEstesa}, \text{Valori}, T), 1))))$

3.4.10 REMOVE

REMOVE elimina un attributo e la corrispondente estensione da una tabella estesa.

Sintassi :

<Rimozione> ::= REMOVE <attributo> IN <tabella_estesa>

Implementazione :

REMOVE(T, A) is

Flocale_EliminaColonna(Fschema_EliminaColonna(T, A))

dove

Flocale_EliminaColonna opera sulla parte Valori della tabella *T* e tupla per tupla, elimina l'elemento presente in posizione *Pos* e l'elemento in posizione *NA+Pos*

Flocale_EliminaColonna(T, Pos, NA) is

if ($\delta(\epsilon(\text{TabEstesa}, \text{Valori}, T), 1) == \text{null}$)

then T

else Aggiungi(Tr_elimina_colonna($\delta(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, T), 1), 1)$, Pos, NA),

Flocale_EliminaColonna(RiduciTabella(T), Pos, NA))

Tr_elimina_colonna opera sulla singola tupla *TP*

Tr_elimina_colonna(TP, Pos, NA) is

$\lambda (\lambda (TP, Pos), NA+Pos)$

3.4.11 NEWATTRIBUTE

Sintassi :

**<NuovoAttributo> ::= NEWATTRIBUTE <stringa> IN <tabella_estesa>
[IS <Espr_NuovoAttributo >]**

<Espr_NuovoAttributo> ::= <Espressione>

| COPY OF <attributo>

<Espressione> ::= (<Operazione> <SeqOperandi>)

<Operazione> ::= qualunque operazione su stringhe o numeri θ

<SeqOperandi> ::= <Operando> , <SeqOperandi> | <Operando>

<Operando> ::= ATTRIBUTE <attributo>

| <Espressione>

| qualunque costante numerica o di tipo stringa c

Descrizione del costrutto :

NEWATTRIBUTE aggiunge alla tabella un nuovo attributo, il cui nome è specificato nel tag *NEWATTRIBUTE* <stringa>. Si noti che la tabella viene anche estesa con il corrispondente campo di estensione per realizzare il meccanismo della marcatura. Per quanto riguarda il valore del nuovo attributo, esso è costruito attraverso la specifica dell'espressione contenuta nel tag *IS* <Espressione>. Se questo tag dovesse mancare, l'attributo, in ogni tupla della tabella, viene inizializzato con il valore vuoto *null*.

L'espressione può essere costituita da un qualunque operatore su stringhe oppure su numeri, applicato ad un numero opportuno di operandi. Gli operandi a loro volta possono essere delle costanti, oppure delle variabili locali alle tuple, oppure possono essere a loro volta delle espressioni. Le variabili sono rappresentate dal tag *ATTRIBUTE*, seguito dal nome di un attributo della tabella; per ogni tupla, esse contengono il valore corrente dell'attributo specificato. Inoltre un'espressione può anche semplicemente indicare che il valore da inserire nel nuovo attributo è la copia di un altro attributo.

Il nuovo attributo non ha alcuna informazione di marcatura associata.

Si osservi anche che non è prevista la costruzione del nuovo attributo attraverso operazioni di riscrittura, ma sarà sufficiente ricorrere al costrutto apposito di riscrittura *RWRULES*.

Sintassi astratta :

NEWATTRIBUTE NomeAttributo IN Tabella

[*IS* Espressione]

≡

NEWATTRIBUTE(Tabella, NomeAttributo, ValoreAttributo)

dove

- ValoreAttributo == *null* se il tag *IS* <Espressione> non è indicato
- ValoreAttributo == *Espressione* se il valore del nuovo attributo è specificato da *Espressione*

Implementazione :

NEWATTRIBUTE(Tabella, Nome, Espressione) is

Flocale_ValutaEspressione(Fschema_AggiungiColonna(Tabella, Nome) ,
Espressione)

dove

Flocale_ValutaEspressione procede tupla per tupla e valuta il valore dell'espressione *Espr*, inserendo il risultato nella ultima colonna, relativa al nuovo attributo con cui la tabella è stata precedentemente estesa. Se *Espr* è uguale a *null*, tale attributo viene inizializzato con il valore *null*.

Flocale_ValutaEspressione(Tab , Espr) is

```

if (δ(ε(TabEstesa,Valori,Tab),1) == null)
  then Tab
  else Aggiungi( Tr_valuta_espressione( δ(δ(ε(TabEstesa,Valori,T),1),1),
                                         δ(ε(TabEstesa,NumAttr,T),1)),
                                         Tab, Espr),
                Flocale_ValutaEspressione ( RiduciTabella(Tab), Espr ) )

```

Tr_valuta_espressione valuta l'espressione *Espr* limitatamente alla tupla *TP*, inserisce il risultato in posizione *NA*, e inserisce il valore *null* in posizione *2*NA*

Tr_valuta_espressione(TP, NA, Tab, Espr) is

```

ζ( null,
  ζ( if (Espr == null) then null
      else SemanticaEspressioneNA(TP, Tab, AlberoSintattico(Espr)),

```

TP, NA)
2*NA)

SemanticaEspressioneNA valuta l'espressione corrispondente all'albero sintattico Espr

SemanticaEspressioneNA(TP, Tab, Espr) is

case (Figli(Espr)) of

{

("COPY OF " <attributo>) :

$\delta (TP , \tau (\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, \text{Tab}), 1),$

$\text{Figlio}(\text{Figlio}(\text{Espr}, 2), 1)))$

(<Espressione>) :

$\text{ValutazioneSemantica}(TP, \text{Tab}, \text{Figlio}(\text{Espr}, 1))$

}

3.4.12 FOLD

La trasformazione di “ripiegamento” attraverso il costrutto FOLD consiste nella riduzione del numero di colonne di una tabella. Tuttavia il contenuto della tupla non va perso in nessuna parte, perché ogni tupla viene duplicata e mantiene invariati tutti gli attributi, esclusi i due attributi coinvolti nella trasformazione. I due corrispondenti valori andranno a finire rispettivamente nella prima e nella seconda copia di ogni tupla. Un esempio aiuta a comprendere immediatamente la semantica della trasformazione.

Sia data la seguente tabella estesa *TE*:

A1	A2	A3	A4	A5	Est1	Est2	Est3	Est4	Est5
<i>x1</i>	<i>x2</i>	<i>x3</i>	<i>x4</i>	<i>x5</i>	<i>e_x1</i>	<i>e_x2</i>	<i>e_x3</i>	<i>e_x4</i>	<i>e_x5</i>
<i>y1</i>	<i>y2</i>	<i>y3</i>	<i>y4</i>	<i>y5</i>
<i>z1</i>	<i>z2</i>	<i>z3</i>	<i>z4</i>	<i>z5</i>					
...			

allora l'espressione

FOLD *A4* **ON** *A2* **IN** *TE*

è equivalente alla tabella estesa *TE'* seguente:

A1	A2	A3	A5	Est1	Est2	Est3	Est5
<i>x1</i>	<i>x2</i>	<i>x3</i>	<i>x5</i>	<i>e_x1</i>	<i>e_x2</i>	<i>e_x3</i>	<i>e_x5</i>
<i>x1</i>	<i>x4</i>	<i>x3</i>	<i>x5</i>	<i>e_x1</i>	<i>e_x4</i>	<i>e_x3</i>	<i>e_x5</i>
<i>y1</i>	<i>y2</i>	<i>y3</i>	<i>y5</i>
<i>y1</i>	<i>y4</i>	<i>y3</i>	<i>y5</i>				
<i>z1</i>	<i>z2</i>	<i>z3</i>	<i>z5</i>				
<i>z1</i>	<i>z4</i>	<i>z3</i>	<i>z5</i>				
...		

Dallo schema della tabella è scomparso il primo dei due attributi, ma il corrispondente valore è stato inserito nella replica di ogni tupla. Lo stesso è avvenuto per quanto riguarda la parte di estensione della tabella.

Si noti che non è stato possibile rispettare lo schema di implementazione delle trasformazioni locali di schema, il quale garantisce la validità di alcune utili proprietà.

Sintassi :

<Ripiegamento> ::= FOLD <attributo> ON <attributo> IN <tabella_estesa>

Implementazione :

FOLD(Tabella, Attributo1, Attributo2) is

$\delta(\text{Fschema_EliminaColonna}(\text{F_fold}(\text{Tabella}, \text{Attributo1}, \text{Attributo2}), \text{Attributo1}), 1)$

F_fold opera solamente sull'elemento Valori dell'oggetto Tab; essa replica ogni tupla, e sostituisce nella seconda copia il valore dell'attributo Att2 con il valore dell'attributo Att1, mantenendo coerenza con la parte di estensione della tabella

F_fold(Tab, Att1, Att2) is

```

if (δ(ε(TabEstesa,Valori,Tab),1) == null)
  then Tab
  else Aggiungi( Tr_fold1( δ(δ(ε(TabEstesa,Valori,Tab),1),1),
                           τ(δ(ε(TabEstesa,Attributi,Tab),1), Att1),
                           δ(ε(TabEstesa,NumAttr,Tab),1)-1),
                 Aggiungi( Tr_fold2( δ(δ(ε(TabEstesa,Valori,Tab),1),1),
                                     τ(δ(ε(TabEstesa,Attributi,Tab),1), Att1),
                                     τ(δ(ε(TabEstesa,Attributi,Tab),1), Att2),
                                     δ(ε(TabEstesa,NumAttr,Tab),1)-1),
                                   F_fold( RiduciTabella(Tab), Att1, Att2 ) ) ) )

```

Tr_fold1 elimina dalla tupla TP gli elementi in posizione Pos1 e in posizione NA+Pos1

Tr_fold1(TP, Pos1, NA) is

```
λ (λ (TP, Pos1), NA+Pos1)
```

Tr_fold2 sposta (rimuovendo) l'elemento che si trova in posizione Pos1 della tupla TP, nella posizione Pos2; successivamente sposta nella posizione NA+Pos2 l'elemento che si trova in posizione NA+Pos1

Tr_fold2(TP, Pos1, Pos2, NA) is

```

λ ( φ ( δ(TP, NA+Pos1),
        λ ( φ ( δ(TP, Pos1), TP, Pos2 ) , Pos1),
        NA+Pos2),
    NA+Pos1)

```

3.4.13 MARK

Si è previsto un secondo costrutto apposito per la marcatura, che si distingue dalla clausola di marcatura *ruleM* per due motivi. Innanzitutto la condizione di marcatura di una tupla è *multiattributo*, e quindi non dipende solamente dal valore di una singola colonna. In secondo luogo il costrutto *MARK* permette di valutare un'espressione piuttosto che effettuare un confronto tra stringhe: la condizione di marcatura è perciò espressa attraverso una operazione di confronto tra due espressioni, oppure attraverso una combinazione, in **AND**, **OR** oppure **NOT**, di tali condizioni primitive. Le operazioni di confronto possono essere $<$, \leq , $>$, \geq e $=$ per valori numerici, e $=$ per stringhe. Si noti che, per la verifica di una condizione di marcatura sul valore di una stringa, è in generale consigliabile utilizzare la regola di marcatura *ruleM*, basata sulle espressioni regolari ed il meccanismo di pattern matching. Essa è infatti molto più flessibile ed ha una maggiore potenza espressiva. L'unico caso in cui non si può fare a meno di utilizzare il costrutto *MARK* è quando almeno una delle due stringhe confrontate per uguaglianza è formata dalla concatenazione di più attributi, oppure quando le due stringhe messe a confronto rappresentano i valori di due attributi diversi.

Nel caso in cui non dovesse essere sufficiente una condizione di semplice uguaglianza tra stringhe, ma fosse richiesto un confronto attraverso pattern matching, e la condizione riguarda più attributi, allora è necessario costruire prima un nuovo attributo tramite *NEWATTRIBUTE*, specificando il valore che esso assume con un'espressione multiattributo, e successivamente applicare la clausola per la marcatura *ruleM* al nuovo attributo così costruito.

Sintassi :

```
<Marcatura> ::= MARK <attributo> IN <tabella_estesa>
                IF ( <Condizione_mark> )
                [ WITH CODE <n> ]
```

```
<Condizione_mark> ::= <Espr_mark> <Operatore_confronto> <Espr_mark>
```

| (<Condizione_mark>) AND (<Condizione_mark>)
 | (<Condizione_mark>) OR (<Condizione_mark>)
 | NOT (<Condizione_mark>)
 <Operatore_confronto> ::= = | < | ≤ | > | ≥
 <Espr_mark> ::= <Espressione>
 | <Operando>
 <Espressione> ::= (<Operazione> <SeqOperandi>)
 <Operazione> ::= *qualunque operazione su stringhe o numeri θ*
 <SeqOperandi> ::= <Operando> , <SeqOperandi> | <Operando>
 <Operando> ::= ATTRIBUTE <attributo>
 | <Espressione>
 | *qualunque costante numerica o di tipo stringa c*

Descrizione del costrutto :

La condizione di marcatura è specificata tramite il tag *IF* (<Condizione_mark>), e nelle tuple per cui vale la condizione viene marcato l'attributo indicato in *MARK* <attributo>. Con il tag opzionale *WITH CODE* <n> è possibile inoltre assegnare un codice di marcatura.

Sintassi astratta :

MARK *Attributo* IN *Tabella*
 IF *Condizione* [WITH CODE *Codice*]

≡

MARK(Tabella, Attributo, Condizione, Marca)


```

then NuovaTupla(TP,  $\delta$ (TP,POS), POS ,
                if (ValutaCondizione(TP, Tab, AlberoSintattico(Cond)))
                    then “mark”+ Cond
                    else null ,
                NA)
else NuovaTupla(TP,  $\delta$ (TP,POS), POS ,
                if (ValutaCondizione(TP, Tab, AlberoSintattico(Cond)))
                    then “mark”+ Cond + ”Codice: ”+M
                    else null ,
                NA)

```

ValutaCondizione restituisce il valore booleano corrispondente alla valutazione, rispetto alla tupla TP, della condizione di marcatura Cond, rappresentata attraverso il suo albero sintattico

ValutaCondizione(TP, Tab, Cond) is

```

case (Figli(Cond)) of
{
  (<Espr_mark> <Operatore_confronto> <Espr_mark>):
    Semantica( SemanticaEsprMark(TP, Tab, Figlio(Cond, 1))
              Semantica(Figlio(Cond, 2))
              SemanticaEsprMark(TP, Tab, Figlio(Cond, 3)) )

  (“<Condizione_mark> ”) AND (“<Condizione_mark> ”) ):
    ValutaCondizione(TP, Tab, Figlio(Cond, 2)) &sem
    ValutaCondizione(TP, Tab, Figlio(Cond, 4))

  (“<Condizione_mark> ”) OR (“<Condizione_mark> ”) ):
    ValutaCondizione(TP, Tab, Figlio(Cond, 2)) ||sem
    ValutaCondizione(TP, Tab, Figlio(Cond, 4))

  (“NOT (“<Condizione_mark> ”) ”) ):
    notsem ValutaCondizione(TP, Tab, Figlio(Cond, 2))

```

}

SemanticaEsprMark valuta l'espressione corrispondente all'albero sintattico Espr

SemanticaEsprMark(TP, Tab, Espr) is

ValutazioneSemantica(TP, Tab, Figlio(Espr,1))

3.4.14 La discretizzazione

La discretizzazione consiste nel ridurre il numero di possibili valori che un attributo può assumere. A questo scopo si divide il dominio in un certo numero di intervalli, e i valori attuali di un attributo sono sostituiti con le etichette di tali intervalli. La riduzione del numero di valori di un attributo è di particolare beneficio per i metodi di classificazione basati su alberi di decisione. Tali metodi sono tipicamente ricorsivi, ed il loro costo computazionale ad ogni passo è per lo più dovuto all'ordinamento dei dati. Risulterà quindi più efficiente ordinare un numero inferiore di valori distinti.

Un esempio di discretizzazione è quello di sostituire un concetto di basso livello, come l'età, con un concetto di livello superiore, identificato dai valori: giovane, di media età, anziano. Nonostante vadano persi con la discretizzazione i dettagli delle informazioni contenute nei dati, i dati generalizzati possono risultare più significativi e facili da interpretare. Inoltre essi occupano spesso uno spazio minore rispetto ai dati originali.

È possibile ripetere il procedimento di discretizzazione più volte, ottenendo così una gerarchia di generalizzazioni, in modo da soddisfare scopi diversi rispetto al livello di dettaglio dei dati.

3.4.14.1 La discretizzazione dei dati numerici

Il metodo di discretizzazione di dati numerici più diffuso è il metodo di *binning*, il quale consiste nel partizionare i dati in un certo numero di intervalli, che può essere

specificato o meno. Le due varianti principali consistono, nel primo caso, nella suddivisione in intervalli della stessa ampiezza, e nel secondo caso, in intervalli tali che, idealmente, ricada lo stesso numero di dati all'interno di ognuno di essi. La prima variante è nota sotto il nome di *Natural Binning*, mentre la seconda è chiamata *Equal Frequency Binning*.

Una volta partizionati i dati è necessario stabilire quali saranno le etichette degli insiemi ottenuti. Nel linguaggio DPL si prevede che le etichette possano essere costituite dalla media aritmetica dei dati, dalla mediana, da uno dei due estremi dell'intervallo, oppure da un valore categorico, come ad esempio il nome di una classe.

Per implementare un metodo di discretizzazione esistono diverse tecniche, tra cui l'*analisi di istogramma*, che serve per partizionare i dati in insiemi della stessa cardinalità, oppure tecniche di *clustering* proprie della fase di data mining. Infine esistono tecniche di discretizzazione *supervisionata*, che verranno discusse nel prossimo paragrafo, tra cui il metodo del *ChiMerge*, e la discretizzazione *basata sulla misurazione dell'entropia*.

3.4.14.2 La discretizzazione dei dati categorici

I metodi di generalizzazione di dati categorici sono molto diversi rispetto alla discretizzazione numerica, in quanto sui dati categorici non è definito un ordinamento. Esempi di tali dati sono le locazioni geografiche, le categorie di impieghi, le classi di oggetti. Alcuni possibili metodi sono i seguenti:

- a) la specificazione di un ordinamento sui possibili valori assunti, e la successiva applicazione di una discretizzazione basata su ordinamento; ad esempio si potrebbe definire un ordinamento sulle locazioni geografiche basato sulle loro coordinate;
- b) la specificazione di una gerarchia attraverso la definizione di gruppi di appartenenza; questa tecnica è applicabile solo quando il numero di possibili

valori assunti in partenza è trattabile, ad esempio partizionando “a mano” le province in gruppi etichettati con la corrispondente regione.

Sintassi dei costrutti di discretizzazione

```

<Discretizzazione> ::= <Discret_num> | <Discret_categ>
<Discret_num> ::= DISCRETIZE (EQUIWIDTH | EQUIDEPTH) <attributo>
                IN <tabella_estesa>
                WITH (WIDTH <n> | DEPTH <n> | <n> INTERVALS)
                SMOOTH BY (MEAN | MEDIAN | INF | SUP |
                <Enumerazione>)
                [EXCEPTION [<n>]]
<Enumerazione> ::= { <stringa> <Enum>}
<Enum> ::= , <stringa> <Enum> | ε
<Discret_categ> ::= SPECIFY ORDERING <tabella> ON <Discret_num>
                | SPECIFY HIERARCHY <tabella> ON
                DISCRETIZE <attributo> IN <tabella_estesa>
                EXCEPTION [<n>]

```

Descrizione del costrutto :

I due tag alternativi *EQUIWIDTH* e *EQUIDEPTH* indicano, rispettivamente, che gli intervalli di partizionamento hanno ampiezza uguale, oppure capacità uguale. Con il tag *WITH WIDTH <n>* si può fissare l'ampiezza costante degli intervalli (uguale a n , tranne per l'ultimo intervallo), mentre con il tag *WITH DEPTH <n>* si può specificare la cardinalità di ogni intervallo, intesa come il numero di valori che ricadono in esso. Tale “profondità” sarà uguale a n per tutti gli intervalli, esclusi eventualmente gli ultimi k intervalli, con $0 \leq k < n$, i quali avranno profondità pari a $n-1$. Infine, il tag *WITH <n>*

INTERVALS indica che l'insieme di partenza viene partizionato in n intervalli, e se esso è associato alla variante *EQUIDDEPTH*, e il numero totale di valori nell'insieme da partizionare è m , allora il partizionamento risultante è quello equivalente alla variante *DISCRETIZE EQUIDDEPTH ... WITH DEPTH $\lceil m/n \rceil$...*.

Le etichette delle partizioni sono definite dal tag *SMOOTH BY etichetta*, dove *etichetta* può indicare la media aritmetica di ogni partizione, la mediana, uno dei due estremi dell'intervallo di partizionamento, oppure un'enumerazione di valori categorici. In quest'ultimo caso, le etichette presenti nell'enumerazione vengono assegnate nello stesso ordine agli intervalli di partizionamento. È necessario specificare il tag *EXCEPTION*, per marcare i valori che non hanno un'etichetta associata, nel caso in cui l'enumerazione di etichette non dovesse coprire tutte le partizioni trovate. In questo caso, l'attributo conserva il valore originario, e la tupla viene "marcata".

Nelle due varianti di discretizzazione di dati categorici, è necessario specificare un ordinamento oppure una gerarchia, in entrambi i casi attraverso una tabella con due soli attributi. Una tabella che definisce un *ordinamento* contiene nella prima colonna i valori categorici che devono essere discretizzati, e nella seconda colonna un valore numerico associato. Una tabella che definisce una *gerarchia* contiene nella prima colonna di nuovo i valori categorici da discretizzare, e nella seconda colonna una generalizzazione del valore contenuto nella prima colonna. In questo modo il partizionamento è definito a priori dall'utente. In entrambe le varianti, se i valori dell'insieme da discretizzare non fossero del tutto coperti dalla prima colonna della tabella di specificazione, allora l'attributo su cui non è definita la generalizzazione mantiene il valore originario, e la tupla a cui esso appartiene viene "marcata" opportunamente.

Alcune restrizioni :

- assieme al tag *EQUIWIDTH* sono ammesse solamente le due varianti *WITH WIDTH $\langle n \rangle$* e *WITH $\langle n \rangle$ INTERVALS*;
- assieme al tag *EQUIDDEPTH* sono ammesse solamente le due varianti *WITH DEPTH $\langle n \rangle$* e *WITH $\langle n \rangle$ INTERVALS*;

- nella discretizzazione numerica (<*Discret_num*>), il tag *EXCEPTION* va specificato se e soltanto se la sostituzione dei valori avviene tramite un'enumerazione di etichette;
- nella discretizzazione categorica (<*Discret_categ*>), il tag *EXCEPTION* va sempre specificato.

Osservazione:

Si noti che la discretizzazione non è in generale una trasformazione di tipo locale, in quanto è necessario conoscere il valore di tutti i dati, prima di partizionare l'insieme in ingresso. Si tratta di una trasformazione locale solamente nella variante di generalizzazione categorica, nella quale viene specificata una gerarchia predefinita, ossia quando il partizionamento è già definito a priori.

Sintassi astratta :

a) Discretizzazione numerica

```
DISCRETIZE Spec1 Attributo IN Tabella
WITH Spec2 Param2
SMOOTH BY Spec3
[EXCEPTION [Marca]]
```

≡

```
DISCRET_NUMERICA(Tabella, Attributo, Spec1, Spec2, Param2, Spec3, Marca)
```

b) Discretizzazione categorica tramite ordinamento

```
SPECIFY ORDERING TabellaSpec ON
DISCRETIZE Spec1 Attributo IN Tabella
```

WITH *Spec2 Param2*
 SMOOTH BY *Spec3*
 EXCEPTION [*Marca*]

≡

DISCRET_CAT_ORDINAMENTO(TabellaSpec, Tabella, Attributo, Spec1, Spec2, Param2, Spec3, Marca)

c) Discretizzazione categorica tramite gerarchia

SPECIFY HIERARCHY *TabellaSpec* ON
 DISCRETIZE *Attributo* IN *Tabella*
 EXCEPTION [*Marca*]

≡

DISCRET_CAT_GERARCHIA(TabellaSpec, Tabella, Attributo, Marca)

dove

- *Marca* == “ null “ se non c’è il tag EXCEPTION;
- *Marca* == “NO_MARK “ se c’è il tag EXCEPTION ma non è specificato un codice di marcatura;
- *Marca* == *Cod* se le eccezioni sono marcate con il codice *Cod*;
- *Spec1* = EQUIWIDTH | EQUIDEPTH ;
- *Spec2* = WIDTH | DEPTH | INTERVALS;
- *Param2* = *n*;
- *Spec3* = MEAN | MEDIAN | INF | SUP | { *Enumerazione* }
 dove *Enumerazione* è una sequenza di stringhe.

Implementazione :

a) Discretizzazione numerica

DISCRET_NUMERICA(Tabella, Attributo, Spec1, Spec2, Param2, Spec3, Marca)

≡

D_numerica(Tabella, Attributo, Spec1, Spec2, Param2, Spec3, Marca)

dove

D_numerica(T, A, Sp1, Sp2, Par2, Sp3, M) is

Etichetta(Partiziona(T, A, Sp1, Sp2, Par2), A, “Partizione ”+A, Sp3, M)

Partiziona(T, A, Sp1, Sp2, Par2) is

if ((Sp1 == *EQUIWITH*) & (Sp2 == *WIDTH*)) then Partiziona1(T, A, Par2)

else if ((Sp1 == *EQUIWITH*) & (Sp2 == *INTERVALS*)) then Partiziona2(T, A, Par2)

else if ((Sp1 == *EQUIDEPTH*) & (Sp2 == *DEPTH*)) then Partiziona3(T, A, Par2)

else if ((Sp1 == *EQUIDEPTH*) & (Sp2 == *INTERVALS*)) then Partiziona4(T, A, Par2)

Partiziona1 effettua la partizione dei valori dell’attributo A nella tabella T, dove gli intervalli di partizione hanno tutti la stessa ampiezza, e l’ampiezza di ogni intervallo è pari a Par2

Partiziona1(T, A, Par2) = TE

dove

- *Lo schema di TE è esteso temporaneamente con l’attributo “Partizione attributo”, dove attributo è quello a cui si applica la discretizzazione:*

$\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, TE), 1) \equiv \alpha(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, T), 1), (“Partizione”+A))$

- *Il valore dell’attributo A di ogni tupla ricade nell’intervallo contenuto nell’attributo “Partizione attributo” della stessa tupla:*

Per ogni tupla \mathbf{t} contenuta nella sequenza:

$$\delta(\epsilon(\text{TabEstesa}, \text{Valori}, TE), 1)$$

l'attributo A di \mathbf{t} :

$$\delta(\mathbf{t}, \tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, TE), 1), A))$$

ricade nell'intervallo:

$$\delta(\mathbf{t}, \tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, TE), 1), \text{"Partizione attributo"}))$$

- *L'attributo "Partizione attributo" di ogni tupla contiene come valore un intervallo che ha ampiezza pari a Par2:*

Per ogni intervallo \mathbf{i} (tranne l'ultimo) contenuto nella sequenza:

$$\rho(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, TE), 1), (\tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, TE), 1), \text{"Partizione attributo"})))$$

$$\text{Ampiezza}(\mathbf{i}) \equiv \text{Par2}$$

Partizona2 effettua la partizione dei valori dell'attributo A nella tabella T , dove gli intervalli di partizione hanno tutti la stessa ampiezza, e il numero totale di intervalli è pari a Par2

$$\mathbf{Partizona2}(T, A, \text{Par2}) = TE$$

dove

- *Lo schema di TE è esteso temporaneamente con l'attributo "Partizione attributo", dove attributo è quello a cui si applica la discretizzazione:*

$$\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, TE), 1) \equiv \alpha(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, T), 1), (\text{"Partizione"} + A))$$

- *Il valore dell'attributo A di ogni tupla ricade nell'intervallo contenuto nell'attributo "Partizione attributo" della stessa tupla:*

Per ogni tupla t contenuta nella sequenza:

$$\delta(\epsilon(\text{TabEstesa}, \text{Valori}, TE), 1)$$

l'attributo A di t :

$$\delta(t, \tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, TE), 1), A))$$

ricade nell'intervallo:

$$\delta(t, \tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, TE), 1), \text{"Partizione attributo"}))$$

- *Il numero di valori distinti dell'attributo "Partizione attributo" della tabella TE, ossia il numero di intervalli di partizionamento, è pari a Par2:*

Data la sequenza s , contenente i valori dell'attributo "Partizione attributo":

$$\rho(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, TE), 1), \\ (\tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, TE), 1), \\ \text{"Partizione attributo"})))$$

$$\text{ValoriDistinti}(s) \equiv \text{Par2}$$

Partiziona3 effettua la partizione dei valori dell'attributo A nella tabella T, dove gli intervalli di partizione hanno tutti la stessa cardinalità, uguale a Par2, tranne eventualmente gli ultimi intervalli, che hanno cardinalità uguale a Par2-1

$$\text{Partiziona3}(T, A, \text{Par2}) = TE$$

dove

- *Lo schema di TE è esteso temporaneamente con l'attributo "Partizione attributo", dove attributo è quello a cui si applica la discretizzazione:*

$$\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, TE), 1) \equiv \alpha(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, T), 1), ("Partizione" + A))$$

- *Il valore dell'attributo A di ogni tupla ricade nell'intervallo contenuto nell'attributo "Partizione attributo" della stessa tupla:*

Per ogni tupla **t** contenuta nella sequenza:

$$\delta(\epsilon(\text{TabEstesa}, \text{Valori}, TE), 1)$$

l'attributo A di **t**:

$$\delta(\mathbf{t}, \tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, TE), 1), A))$$

ricade nell'intervallo:

$$\delta(\mathbf{t}, \tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, TE), 1), "Partizione attributo"))$$

- *Gli intervalli di partizione hanno tutti la stessa cardinalità, uguale a Par2, tranne eventualmente gli ultimi intervalli, che hanno cardinalità uguale a Par2-1:*

Per ogni intervallo **i** contenuto nella sequenza **s**:

$$\rho(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, TE), 1), \\ (\tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, TE), 1), \\ "Partizione attributo"))))$$

la sequenza **p** di tuple, i cui valori dell'attributo A ricadono in tale intervallo **i**:

p(i)

=

$$\sigma(\text{contenuto in } \mathbf{i}, \tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, TE), 1), A), \delta(\epsilon(\text{TabEstesa}, \text{Valori}, TE), 1))$$

a)

contiene un numero di elementi pari a $Par2$:

$$\text{Cardinalità}(\mathbf{p}) \equiv Par2$$

OPPURE

b)

contiene un numero di elementi pari a $Par2-1$:

$$\text{Cardinalità}(\mathbf{p}) \equiv Par2-1$$

ed esistono k intervalli contenuti in \mathbf{s} , con $0 \leq k < n-1$, tali che $i < i_1 < i_2 < \dots < i_k$,
che hanno tutti cardinalità pari a $Par2-1$:

$$\text{Cardinalità}(\mathbf{p}(i_1)) \equiv \dots \equiv \text{Cardinalità}(\mathbf{p}(i_k)) \equiv Par2-1$$

mentre ogni intervallo $i_j < i$ ha cardinalità uguale a $Par2$:

$$\text{Cardinalità}(\mathbf{p}(i_j)) \equiv Par2$$

Partiziona4 effettua la partizione dei valori dell'attributo A nella tabella T , dove gli intervalli di partizione hanno tutti la stessa cardinalità, tranne eventualmente gli ultimi intervalli, e il numero totale di intervalli è pari a $Par2$

Partiziona4(T, A, Par2) is

$$\text{Partiziona3}(T, A, \lceil \text{Cardinalità}(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, TE), 1)) / Par2 \rceil)$$

Etichetta(T, A, Partiz, Sp3, Enum, M) is

if (Sp3 == *MEAN*) then Etichetta1(*mediaAritmetica*, T, A, Partiz)

else if (Sp3 == *MEDIAN*) then Etichetta1(*moda*, T, A, Partiz)

else if (Sp3 == *INF*) then Etichetta1(*inf*, T, A, Partiz)

else if (Sp3 == *SUP*) then Etichetta1(*sup*, T, A, Partiz)

else if (Sp3 == {<enumerazione>}) then Etichetta2(T, A, Partiz, Enum, M)

Etichetta1 sostituisce ogni valore dell'attributo *A* in *T*, con l'etichetta *Et* della partizione in cui esso ricade; tale partizione è identificata dall'attributo di estensione temporaneo *Partiz*

Etichetta1(Et, T, A, Partiz) is

EliminaSingolaColonna(F_etichetta1(Et, T, A, Partiz), Partiz)

EliminaSingolaColonna elimina l'attributo *A* dallo schema relazionale della tabella estesa *T*

EliminaSingolaColonna(T, A) is

$\gamma(\text{TabEstesa}, ((\mu(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, T), 1), A),$
 $\delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, T), 1) - 1,$
 $\text{EliminaSingoloAttributo}(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, T), 1),$
 $\tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, T), 1), A))$
 $)))$

EliminaSingoloAttributo elimina dalle tuple estese contenute nella sequenza *SV*, l'attributo in posizione *POS*, e restituisce la sequenza modificata *SV*

EliminaSingoloAttributo(SV, POS, NA) is

if (SV == null) then null
 else $\alpha ((\lambda(\delta(\text{SV}, 1), \text{POS}),$
 $\text{EliminaAttributo}(\lambda(\text{SV}, 1), \text{POS}))$

F_etichetta1 effettua la sostituzione dell'attributo *A* in *T*, con l'etichetta *Et* della partizione in cui esso ricade, e restituisce la tabella così modificata *T*

F_etichetta1(Et, T, A, Partiz) is

if ($\delta(\epsilon(\text{TabEstesa}, \text{Valori}, T), 1) == \text{null}$)
 then *T*
 else $\text{Aggiungi}(\text{Tr_etichetta1}(\delta(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, T), 1), 1),$
 $\tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, T), 1), A),$
 $\tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, T), 1), \text{Partiz}),$

$$T, Et)$$

$$F_etichetta1(Et, RiduciTabella(T), A, Partiz))$$

Tr_etichetta1 effettua la sostituzione nella tupla TP

Tr_etichetta1(TP, PosA, PosPartiz, T, Et) is

if (Et == *mediaAritmetica*) then

NuovaTupla(TP, TrovaMediaPartiz(T, δ (TP,PosPartiz), PosA, PosPartiz),
PosA , null, 0)

else if (Et == *moda*) then

NuovaTupla(TP, TrovaModaPartiz(T, δ (TP,PosPartiz), PosA, PosPartiz),
PosA , null, 0)

else if (Et == *inf*) then

NuovaTupla(TP, TrovaInfPartiz(T, δ (TP,PosPartiz), PosA, PosPartiz), PosA
, null, 0)

else if (Et == *sup*) then

NuovaTupla(TP, TrovaSupPartiz(T, δ (TP,PosPartiz), PosA, PosPartiz), PosA
, null, 0)

Restituisce la media aritmetica dell'attributo in posizione PosA, ristretta alle tuple di T che hanno valore uguale a ValPartiz in posizione PosPartiz:

TrovaMediaPartiz(T, ValPartiz, PosA, PosPartiz) is

MediaAritmetica(ρ (σ (== ValPartiz, PosPartiz , δ (ϵ (TabEstesa,Valori, T),1),
(PosA)))

Restituisce la moda dell'attributo in posizione PosA, ristretta alle tuple di T che hanno valore uguale a ValPartiz in posizione PosPartiz:

TrovaModaPartiz(T, ValPartiz, PosA, PosPartiz) is

Moda(ρ (σ (== ValPartiz, PosPartiz , δ (ϵ (TabEstesa,Valori, T),1),
(PosA)))

TrovaInfPartiz(T, ValPartiz, PosA, PosPartiz) is

EstremoInferiore($\rho(\sigma(== ValPartiz, PosPartiz, \delta(\epsilon(TabEstesa, Valori, T), 1), (PosA))$))

TrovaModaPartiz(T, ValPartiz, PosA, PosPartiz) is

EstremoSuperiore($\rho(\sigma(== ValPartiz, PosPartiz, \delta(\epsilon(TabEstesa, Valori, T), 1), (PosA))$))

□

Etichetta2 sostituisce ogni valore dell'attributo *A* in *T*, con l'etichetta della partizione in cui esso ricade; tale partizione è identificata dall'attributo di estensione temporaneo *Partiz*, e le etichette sono contenute nella sequenza *Enum*

Etichetta2(T, A, Partiz, Enum, M) is

EliminaSingolaColonna(F_etichetta2(T, A, Partiz, Enum, M), Partiz)

F_etichetta2(T, A, Partiz, Enum, M) is

if ($\delta(\epsilon(TabEstesa, Valori, T), 1) == \text{null}$)

then T

else Aggiungi(Tr_etichetta2($\delta(\delta(\epsilon(TabEstesa, Valori, T), 1), 1)$,

$\tau(\delta(\epsilon(TabEstesa, Attributi, T), 1), A)$,

$\tau(\delta(\epsilon(TabEstesa, Attributi, T), 1), Partiz)$,

$\delta(\epsilon(TabEstesa, NumAttr, T), 1)$,

T, Enum, M)

F_etichetta2(RiduciTabella(T), A, Partiz, Enum, M))

Tr_etichetta2 effettua la sostituzione nella tupla *TP*

Tr_etichetta2(TP, PosA, PosPartiz, NA, T, Enum, M) is

```

NuovaTupla(TP,  $\delta$ (TrovaEtichetta(T, TP, PosPartiz, PosA, Enum),1),
    PosA ,
    if ( $\delta$ (TrovaEtichetta(T, TP, PosPartiz, PosA, Enum),2) )
        then if (M == NO_MARK)
            then “discret_numerica” +  $\delta$ (TP,PosA)
            else “discret_numerica” +  $\delta$ (TP,PosA) + “Codice: ”+ M
        else null ,
    NA)

```

TrovaEtichetta restituisce una coppia formata dall'etichetta di sostituzione, e da un valore booleano che indica l'esistenza di un'etichetta per la tupla TP

TrovaEtichetta(T, TP, PosPartiz, PosA, Enum) = (etichetta, eccezione)

dove,

per ogni tupla **t** della tabella *T*, contenuta nella sequenza:

$\delta(\epsilon(\text{TabEstesa}, \text{Valori}, T), 1)$

sia **i(t)** il suo intervallo di partizionamento:

$\delta(\mathbf{t}, \text{PosPartiz})$

e sia **j** l'indice di ordinamento di tale intervallo, su un numero complessivo di intervalli uguale a *k*:

$i_1 < i_2 < \dots < i_j < \dots < i_k$

allora,

se esiste l'elemento **j**-esimo nella sequenza *Enum*,

etichetta $\equiv \delta(\text{Enum}, \mathbf{j})$ e

eccezione $\equiv \text{false}$

altrimenti l'attributo mantiene il suo valore originario:

etichetta $\equiv \delta(\mathbf{t}, \text{PosA})$ e

eccezione \equiv *true*

▪

Implementazione :

b) Discretizzazione categorica tramite ordinamento

**DISCRET_CAT_ORDINAMENTO(TabellaSpec, Tabella, Attributo, Spec1,
Spec2, Param2, Spec3, Marca)**

\equiv

D_cat_ord(TabellaSpec, Tabella, Attributo, Spec1, Spec2, Param2, Spec3, Marca)

dove

D_cat_ord(TabellaSpec, Tabella, Attributo, Spec1, Spec2, Param2, Spec3, Marca)

is

**D_numerica(ApplicaSpecifica(Tabella, TabellaSpec, Attributo, Marca),
Attributo, Spec1, Spec2, Param2, Spec3, Marca)**

ApplicaSpecifica sostituisce nella tabella *T* i valori categorici dell'attributo *A*, con i corrispondenti valori numerici, presenti nella tabella di specifica *TSpec*; se un valore categorico non fosse specificato in *TSpec*, allora la tupla corrispondente verrà marcata

ApplicaSpecifica(T, TSpec, A, M) is

if ($\delta(\epsilon(\text{TabEstesa}, \text{Valori}, T), 1) == \text{null}$)

then T

else Aggiungi(Tr_applicaSpecifica($\delta(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, T), 1), 1)$,

$\tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, T), 1), A)$,

$\delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, T), 1)$,

TSpec, M),

ApplicaSpecifica(RiduciTabella(T), TSpec, A, M))

Tr_applicaSpecifica sostituisce il valore numerico nella tupla TP

Tr_applicaSpecifica(TP, PosA, NA, TSpec, M) is

```

NuovaTupla(TP,  $\delta$ (TrovaValoreSpec( $\delta$ (TP, PosA), TSpec, 1),1),
           PosA ,
           if ( not  $\delta$ (TrovaValoreSpec ( $\delta$ (TP, PosA), Tspec, 1),2) )
             then if (M == NO_MARK)
               then “discret_categ_ord” +  $\delta$ (TP,PosA)
               else “discret_categ_ord” +  $\delta$ (TP,PosA) + “Codice: ”+ M
             else null ,
           NA)

```

TrovaValoreSpec cerca nella tabella di specifica TSpec (di tipo Tabella!), a partire dalla tupla I-esima, il valore numerico corrispondente al valore categorico ValCat; restituisce una coppia formata dal valore numerico di specifica, e da un valore booleano, che è vero se e solo se ValCat è presente nella tabella TSpec a partire dalla tupla I-esima

TrovaValoreSpec(ValCat, Tspec, I) is

```

if ( $\delta$ ( $\epsilon$ (Tabella,Valori,TSpec), 1) == null)
  then (ValCat, false)
else if ( $\delta$ (  $\delta$ ( $\epsilon$ (Tabella,Valori,TSpec), 1) , I) == null)
  then (ValCat, false)
else if ( $\delta$ ( $\delta$ (  $\delta$ ( $\epsilon$ (Tabella,Valori,TSpec), 1) , I) , 1) == ValCat)
  then ( $\delta$ ( $\delta$ (  $\delta$ ( $\epsilon$ (Tabella,Valori,TSpec), 1) , I) , 2) , true)
else TrovaValoreSpec(ValCat, Tspec, I+1)

```

▪

Implementazione :

c) Discretizzazione categorica tramite gerarchia**DISCRET_CAT_GERARCHIA(TabellaSpec, Tabella, Attributo, Marca)⁴**

≡

D_cat_ger(TabellaSpec, Tabella, Attributo, Marca)

dove

D_cat_ger(TSpec, T, A, M) is

ApplicaSpecifica(T, TSpec, A, M)

3.4.14.3 La discretizzazione supervisionata

È importante tenere in considerazione una tecnica particolare di discretizzazione, soprattutto quando il contesto di preparazione dei dati è quello di data mining, ovvero di estrazione di conoscenza dai dati. Questa classe particolare di discretizzazione è chiamata *supervisionata*, rispetto alle varianti viste fino ad ora, che appartengono tutte alla classe di *discretizzazione non supervisionata*. Infatti, come si è visto, sono fissati a priori le dimensioni delle partizioni, come ad esempio l'ampiezza, la cardinalità, o il numero degli intervalli di partizionamento. Quello che caratterizza invece la discretizzazione supervisionata è che tali misure non sono fissate dall'utente, ma vengono stabilite, sia in base alla distribuzione dei dati, che in relazione ad una *classificazione* predefinita dei dati. Rispetto agli esempi visti fino ad ora, è quindi necessario specificare delle informazioni di classe.

Il motivo per cui questa tecnica è preferibile in molti casi, è che, tenendo in considerazione la classe di appartenenza di un dato, sia numerico che categorico, quest'ultimo viene discretizzato con un valore relativo alla classe. Se pensiamo alla

⁴ Si noti che solo quest'ultima variante *c)* di discretizzazione è una trasformazione locale.

discretizzazione in ottica relazionale, discretizzare in modo supervisionato un attributo, che chiameremo attributo *obiettivo*, consiste inizialmente nel partizionare le tuple della tabella in base ad un secondo attributo, che chiameremo *di classe*, e successivamente, nella sostituzione di ogni valore dell'attributo obiettivo, con il valore discretizzato della corrispondente partizione. Si noti come anche in questo caso è utile l'integrazione di un linguaggio di interrogazioni di data mining, con un linguaggio di preparazione dei dati. Infatti l'attributo *di classe*, invece di essere un semplice attributo di tipo categorico con valore proveniente dalla classica immissione “manuale” nella base di dati, potrebbe essere invece un attributo di classificazione, con cui la tabella è stata estesa applicando un algoritmo di classificazione, proprio della fase di data mining.

Questa tecnica contribuisce ad ottenere risultati più interessanti ed accurati quando, ai dati così preparati, verranno applicati algoritmi di data mining.

Vediamo ora due possibili varianti di discretizzazione supervisionata.

1. Il metodo del ChiMerge

La discretizzazione supervisionata con il *metodo del ChiMerge* segue una strategia bottom-up. Inizialmente ogni valore è un intervallo a sé, ed in modo iterativo, due intervalli adiacenti sono uniti se sono “simili”, in base ad un attributo *di classe*. Il criterio di similitudine è basato sul test del Chi quadro (χ^2), che è definito nel modo seguente:

siano

- k il numero di valori differenti dell'attributo *di classe*
- A_{ij} il numero di casi della j -esima classe nell' i -esimo intervallo
- R_i numero di casi nell' i -esimo intervallo

$$R_i = \sum_{j=1}^k A_{ij}$$

- C_j il numero di casi nella j -esima classe

$$C_j = \sum_{i=1}^2 A_{ij}$$

- E_{ij} la frequenza attesa di A_{ij} ($A_{ij} = R_i * C_j / N$, dove $N = \sum_i \sum_j A_{ij}$)

Allora il test del *Chi quadro* consiste nel calcolare per ogni coppia di intervalli il valore di χ^2 , dove

$$\chi^2 = \sum_{i=1}^2 \sum_{j=1}^k \frac{(A_{ij} - E_{ij})^2}{E_{ij}}$$

I due intervalli più “simili”, ossia quelli che hanno il valore di χ^2 minimo, vengono uniti, ammesso che tale valore superi una soglia minima prefissata δ . Questa soglia determina il criterio di arresto dell’intero procedimento.

Osservazione:

Si noti che questo metodo riguarda solamente la fase di partizionamento dei dati, mentre l’etichettamento dei dati è una questione affrontata a parte.

2. Il metodo di discretizzazione basata sull’entropia

Anche la discretizzazione *basata sulla misurazione dell’entropia*, come il metodo del *ChiMerge*, affronta solamente il problema di un partizionamento opportuno dei dati,

mentre lascia aperta la questione della scelta delle etichette, con cui i valori dei dati da discretizzare devono essere sostituiti.

Verrà ora descritto il funzionamento di questo metodo, senza darne una semantica operativa formale.

Sia dato un insieme A di dati da discretizzare:

1. Ogni valore di A può essere considerato un potenziale estremo di intervallo oppure separatore T . Ad esempio, un valore v di A può partizionare i dati in due insiemi $A1$ ed $A2$, in modo che, ogni dato di $A1$ sia inferiore a v , e ogni dato di $A2$ sia superiore o uguale a v . In questo modo si è ottenuta una partizione binaria di A .
2. Il valore separatore T di un insieme viene scelto in modo da massimizzare il *beneficio di informazione* risultante dal corrispondente partizionamento. Il beneficio di informazione è dato da

$$I(A,T) = (|A1| / |A| * Ent(A1)) + (|A2| / |A| * Ent(A2)),$$

dove $A1$ e $A2$ sono i due insiemi ottenuti dal partizionamento binario di A in base al valore di T . La funzione *Ent* misura l'entropia di un insieme, la quale viene calcolata in base alla distribuzione dei dati presenti nell'insieme, rispetto a delle classi predefinite. Siano ad esempio date m classi, allora l'entropia di $A1$ è

$$Ent(A1) = -\sum_{(i=1 \dots m)} p_i * \log_2 p_i ,$$

dove p_i indica la probabilità della classe i nell'insieme $A1$, che è determinata dividendo il numero di dati della classe i che si trovano in $A1$, per il numero totale di dati in $A1$. Analogamente si calcola $Ent(A2)$.

3. Per ogni partizionamento ottenuto si applica ricorsivamente la ricerca di un valore separatore T , fino a che un predefinito criterio di arresto non termina il procedimento, come ad esempio la condizione

$$\text{Ent}(A) - I(A,T) > \delta .$$

Sintassi dei costrutti di discretizzazione supervisionata

```

<Discretizzazione> ::= ... | <Discret_supervis>
<Discret_supervis> ::= DISCRETIZE <attributo> IN <tabella_estesa>
                        SUPERVISED WITH <Metodo_Discr_Supervis>
                        USING CLASS <attributo> STOP AT <reale>
                        SMOOTH BY (MEAN | MEDIAN | INF | SUP |
                        <Enumerazione>)
                        [EXCEPTION [<n>]]5
<Metodo_Discr_Supervis> ::= CHI2 | ENTROPY
<Enumerazione> ::= { <stringa> <Enum>}
<Enum> ::= , <stringa> <Enum> | ε

```

3.4.14.4 La scelta del tipo di discretizzazione

Non è in generale semplice decidere quale è la discretizzazione migliore da applicare ad un insieme di dati. Questa scelta è inoltre tanto più cruciale, quanto più intensamente i dati in questione sono utilizzati nella fase di data mining, per estrarre da

⁵ *Attenzione:* Come nella discretizzazione numerica (<Discret_num>), il tag EXCEPTION va specificato se e soltanto se la sostituzione dei valori avviene tramite un'enumerazione di etichette.

essi dei modelli di conoscenza. Si noti infatti che la discretizzazione consiste in una generalizzazione di un concetto, in quanto i valori dei dati vengono sostituiti con nuovi valori meno dettagliati. È allora inevitabile una certa perdita di informazione, dovuta già solo al fatto che il numero di valori distinti delle etichette è inferiore al numero di valori distinti dei dati originali.

È in generale sempre desiderabile ridurre al minimo la perdita di informazione contenuta nei dati. Si può infatti intuire come, in particolar modo i processi di classificazione, possano trarre vantaggio ed ottenere risultati più accurati, quando i dati sono stati precedentemente discretizzati attraverso un metodo supervisionato. Altrettanto comprensibile è il fatto che una discretizzazione non supervisionata, in cui è stata fissata l'ampiezza degli intervalli, possa essere in certi casi non solo dannosa dal punto di vista della perdita di informazione, ma addirittura inutile in quanto processo di generalizzazione. Basti pensare al semplice caso in cui i dati numerici {1,251; 2,05; 3,992; 4} vengano discretizzati attraverso intervalli di ampiezza 1, ottenendo come risultato l'insieme {1; 2; 3; 4}, che ha lo stesso numero di valori distinti.

Vale comunque la pena notare che i metodi supervisionati hanno un certo peso computazionale, e a seconda degli obiettivi, non è sempre necessario applicare questi metodi nella preparazione dei dati. Se si tratta ad esempio di uniformare dati provenienti da sorgenti diverse, aventi formati non uniformi, ha senso discretizzare un attributo fissando l'ampiezza degli intervalli di partizionamento, oppure applicando una discretizzazione categorica attraverso una tabella di specializzazione gerarchica.

Alcuni altri casi, in cui un metodo semplice ed economico come il *Natural Binning* può essere preferibile ad un metodo supervisionato, sono quando i dati risultanti non devono essere sottoposti ad estrazione di conoscenza, oppure quando la distribuzione dei dati è uniforme sul dominio. Non si dimentichi che è sempre importante analizzare i dati da *preprocessare*, ed in questo caso, una statistica sulla distribuzione, come potrebbe essere un semplice istogramma, aiuta nella scelta del metodo di discretizzazione.

Per quanto riguarda la scelta dei parametri di discretizzazione non supervisionata, riportiamo alcune informazioni, senza entrare nei dettagli di tali risultati:

- Il numero ottimale C di partizioni è funzione del numero N di elementi (Sturges, 1929)

$$C = 1 + \frac{10}{3} \log_{10}(N)$$

- L'ampiezza ottimale degli intervalli di partizionamento dipende dalla varianza e dal numero dei dati (Scott, 1979)

$$h = \frac{3,5 \cdot s}{\sqrt{N}}$$

3.4.15 La normalizzazione

La normalizzazione consiste nel ridimensionare i valori di un attributo in modo che ricadano tutti in un intervallo specificato, ad esempio tra -1.0 e 1.0, oppure tra 0.0 e 1.0. Questa trasformazione è particolarmente utile negli algoritmi di classificazione delle reti neurali, oppure negli algoritmi di classificazione e di clustering di tipo “nearest neighbor”, dove è richiesta la misura di distanze. Utilizzando infatti l'algoritmo di classificazione con propagazione all'indietro nelle reti neurali, la normalizzazione degli attributi misurati nei campioni di training contribuisce a velocizzare la fase di apprendimento. Invece nei metodi di tipo “nearest neighbor” che si basano sulla misura di distanze, la normalizzazione serve a calibrare le misure effettuate su attributi diversi, ossia su dimensioni diverse. Altrimenti le distanze calcolate su un attributo con valori binari avrebbero un peso minimo rispetto alle misure effettuate su un attributo con valori molto alti, come potrebbe essere ad esempio l'attributo stipendio.

Ci sono molti metodi per normalizzare i dati; i tre principali sono i seguenti:

- *Normalizzazione minimo-massimo*
- *Normalizzazione basata sulla media*
- *Normalizzazione decimale*

La **normalizzazione minimo-massimo** è effettuata attraverso la formula

$$v' = (v - \min_A / \max_A - \min_A) * (\text{nuovo_max}_A - \text{nuovo_min}_A) + \text{nuovo_min}_A .$$

e calcola i nuovi valori v' dell'attributo A a partire da quelli originali v con una trasformazione lineare, mantenendo inalterato il rapporto tra di essi. Questi nuovi valori verranno a ricadere nel nuovo intervallo $[\text{nuovo_min}_A, \text{nuovo_max}_A]$ ricoprendolo esattamente.

Con la **normalizzazione basata sulla media** i valori di un attributo A sono calcolati in base alla media ed alla deviazione standard di A . La formula di trasformazione è la seguente:

$$v' = (v - \text{media}_A) / \text{dev_standard}_A .$$

Questo metodo è di particolare utilità quando non sono noti il valore massimo ed il valore minimo attuali, oppure in presenza di *outliers* che avrebbero un impatto troppo forte sulla normalizzazione minimo-massimo.

Nella **normalizzazione decimale** viene spostata la virgola decimale dei valori di A in modo che

$$v' = v/10^j \quad \text{e } j \text{ sia il più piccolo intero tale che } \text{Max}(|v'|) < 1.$$

Tutte le tre varianti di normalizzazione sono funzioni non locali, e richiedono perciò tutti i valori dell'attributo A prima di effettuare la trasformazione. Tuttavia nei primi due casi si potrebbero sfruttare i metadati e trasformare direttamente ogni tupla della tabella. Il valore minimo ed il valore massimo potrebbero inoltre essere trovati immediatamente se la tabella è ordinata sull'attributo A , oppure eseguendo una interrogazione con il calcolo del valore aggregato richiesto (min, max, media, deviazione standard).

In ogni caso sarebbe opportuno memorizzare i parametri della normalizzazione in modo che dati futuri siano trasformati in modo uniforme.

Sintassi dei costrutti di normalizzazione

```

<Normalizzazione> ::= NORMALIZE_<Metodo> ( <Seq_attributi> )
                        IN <tabella_estesa> [ EXCEPTION [ <codice> ] ]
<Metodo> ::= MIN_MAX <real> <real> | Z_SCORE | DECIMAL
            | MIN_MAX_L <real> <real> <real> <real> | Z_SCORE_L <real>
<real>
<Seq_attributi> ::= <attributo> [, <Seq_attributi>] | <attributo>

```

Descrizione del costrutto :

A seconda del metodo di normalizzazione che si intende applicare, si usa il costrutto opportuno; le varianti *NORMALIZE_MIN_MAX_L* e *NORMALIZE_Z_SCORE_L* sono delle trasformazioni locali, perché non richiedono l'intera tabella prima di procedere a trasformare le tuple. Affinché ciò sia possibile, il programmatore deve conoscere e specificare nel primo caso il valore minimo attuale e il valore massimo attuale, e nel secondo caso la media aritmetica e la deviazione standard rispetto all'attributo da normalizzare. Il primo dei due è l'unico costrutto che ammette il tag *EXCEPTION*, per marcare le eccezioni sollevate quando il minimo e massimo attuali non sono corretti, e succede che un valore normalizzato ricade al di fuori del nuovo intervallo specificato.

L'utilità delle varianti locali sta innanzitutto nel fatto che godono di tutte le proprietà delle funzioni locali, e quindi si prestano ad essere ottimizzate meglio. Esse inoltre rendono possibile normalizzare in modo uniforme un nuovo insieme di dati, immesso in un secondo momento, rispetto ai dati già normalizzati prima.

Sintassi astratta :

NORMALIZE_m [R1] [R2] [R3] [R4] (Att1,Att2,...) IN Te

[EXCEPTION [Cod]]

≡ NORM_m ([R1], [R2], [R3], [R4], (Att1,Att2,...), Te , Mark)

dove

- Mark == “ null“ se non c’è il tag EXCEPTION;
- Mark == “NO_MARK“ se c’è il tag EXCEPTION ma non è specificato un codice di marcatura;
- Mark == Cod se le eccezioni sono marcate con il codice *Cod*;
- m indica il metodo di normalizzazione usato;
- gli attributi da normalizzare *Att1, Att2, ...* sono tutti diversi.

Implementazione :

1) NORM_MIN_MAX (min, max, seqAttr, tabella)

≡

Fnorm_MM(min, max, seqAttr, tabella)

dove

Fnorm_MM (MIN, MAX, ATTRS, TE) is

if (ATTRS == null) then TE

else Fnorm1_MM(TrovaMinAttuale(TE, δ(ATTRS,1)),

TrovaMaxAttuale(TE, δ(ATTRS,1)),

MIN, MAX, δ(ATTRS,1),

Fnorm_MM(MIN, MAX, λ (ATTRS,1), TE) ,

null)

TrovaMinAttuale(TE, A) is

if (TE == null) then null

else *min*($\delta(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, \text{TE}), 1), 1)$,

$\tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, \text{TE}), 1), \text{A})$),

TrovaMinAttuale (RiduciTabella(TE), A))

TrovaMaxAttuale(TE, A) is

if (TE == null) then null

else *max*($\delta(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, \text{TE}), 1), 1)$,

$\tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, \text{TE}), 1), \text{A})$),

TrovaMaxAttuale (RiduciTabella(TE), A))

Fnorm1_MM (MIN_ATT, MAX_ATT, MIN, MAX, ATT, TE, M) is

if ((MIN_ATT == null) OR (MAX_ATT == null)

OR ($\delta(\epsilon(\text{TabEstesa}, \text{Valori}, \text{TE}), 1) == \text{null}$))

then TE

else Aggiungi (Tr_norm_MM($\delta(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, \text{TE}), 1), 1)$,

MIN_ATT, MAX_ATT, MIN, MAX,

$\tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, \text{TE}), 1), \text{ATT})$,

$\delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, \text{TE}), 1)$,

M),

Fnorm1_MM(MIN_ATT, MAX_ATT, MIN, MAX, ATT,

RiduciTabella(TE), M))

Tr_norm_MM (TP, MIN_ATT, MAX_ATT, MIN, MAX, POS, NA, M) is

if (M == null)

then NuovaTupla(TP,

$(\delta(\text{TP}, \text{POS}) - \text{MIN_ATT} / \text{MAX_ATT} - \text{MIN_ATT}) * (\text{MAX} - \text{MIN}) + \text{MIN}$,

POS , null , NA)

else if (M == "NO_MARK")

then NuovaTupla(TP,

```

(δ(TP,POS) – MIN_ATT / MAX_ATT – MIN_ATT) * (MAX – MIN) + MIN,
POS,
if Ecc
    then “normMML”+ MIN_ATT + MAX_ATT + MIN + MAX + δ(TP,POS)
    else null ,
NA)
else NuovaTupla(TP,
(δ(TP,POS) – MIN_ATT / MAX_ATT – MIN_ATT) * (MAX – MIN) + MIN,
POS,
if Ecc
    then “normMML”+ MIN_ATT + MAX_ATT + MIN + MAX +
        δ(TP,POS) + M
    else null ,
NA)

```

2) NORM_MIN_MAX_L(minAtt, maxAtt, min, max, seqAttr, tabella, marca)

≡

Fnorm_MM_L (minAtt, maxAtt, min, max, seqAttr, tabella, marca)

dove

Fnorm_MM_L (MIN_ATT, MAX_ATT, MIN, MAX, ATTRS, TE, M) is

if (ATTRS == null) then TE

else Fnorm1_MM(MIN_ATT,

MAX_ATT,

MIN, MAX, δ(ATTRS,1),

Fnorm_MM_L(MIN_ATT, MAX_ATT, MIN,

MAX, λ (ATTRS,1), TE, M) , M)

3) NORM_Z_SCORE (seqAttr, tabella)

≡

Fnorm_Z (seqAttr, tabella)

dove

Fnorm_Z (ATTRS, TE) is

if (ATTRS == null) then TE

else Fnorm1_Z (TrovaMedia(TE, δ (ATTRS,1)),
TrovaDevStd(TE, δ (ATTRS,1)),
 δ (ATTRS,1),
Fnorm_Z (λ (ATTRS,1), TE)

TrovaMedia(TE, A) = m

dove

m è la media aritmetica dei valori corrispondenti all'attributo *A* nella tabella *TE*,
ossia è la media aritmetica dei valori contenuti nella sequenza
 $\rho(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, \text{TE}), 1), (\tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, \text{TE}), 1), A)))$.

TrovaDevStd(TE, A) = s

dove

s è la deviazione standard dei valori corrispondenti all'attributo *A* nella tabella *TE*,
che sono contenuti nella sequenza
 $\rho(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, \text{TE}), 1), (\tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, \text{TE}), 1), A)))$.

Fnorm1_Z (MED, DEV, ATT, TE) isif ((MED == null) OR (DEV == null) OR ($\delta(\epsilon(\text{TabEstesa}, \text{Valori}, \text{TE}), 1) == \text{null}$))

then TE

else Aggiungi (Tr_norm_Z($\delta(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, \text{TE}), 1), 1)$,

$$\begin{aligned} & \text{MED, DEV,} \\ & \tau(\delta(\varepsilon(\text{TabEstesa,Attributi,TE}),1), \text{ATT}), \\ & \delta(\varepsilon(\text{TabEstesa,NumAttr,TE}),1)), \\ & \text{Fnorm1_Z}(\text{MED, DEV, ATT, RiduciTabella(TE)}) \end{aligned}$$

Tr_norm_Z (TP, MED, DEV, POS, NA) is

NuovaTupla(TP,
 $(\delta(\text{TP,POS}) - \text{MED}) / \text{DEV}$,
 POS , null , NA)

4) NORM_Z_SCORE_L(media, devStd, seqAttr, tabella)

≡

Fnorm_Z_L (media, devStd, seqAttr, tabella)

dove

Fnorm_Z_L (MED, DEV, ATTRS, TE) is

if (ATTRS == null) then TE

$$\begin{aligned} & \text{else Fnorm1_Z (MED,} \\ & \text{DEV,} \\ & \delta(\text{ATTRS},1), \\ & \text{Fnorm_Z (MED, DEV, } \lambda (\text{ATTRS},1), \text{TE))} \end{aligned}$$

5) NORM_DECIMAL (seqAttr, tabella)

≡

Fnorm_dec (seqAttr, tabella)

dove

Fnorm_dec (ATTRS, TE) is

```
if (ATTRS == null)
  then TE
  else Fnorm1_dec ( TrovaEsponente(TE,  $\delta$ (ATTRS,1)),
                   $\delta$ (ATTRS,1),
                  Fnorm_dec ( $\lambda$  (ATTRS,1), TE) )
```

TrovaEsponente (TE, A) = j

con **j** il piú piccolo intero t.c.

$$\text{Max}(|v'|) < 1$$

dove

per ogni v appartenente alla sequenza

$$\rho(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, \text{TE}), 1), (\tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, \text{TE}), 1), A)))$$

(ossia per ogni valore v dell'attributo A nella tabella TE),

i valori v' sono calcolati con la formula $v' = v/10^j$.

Fnorm1_dec (ESP, ATT, TE) is

```
if ( $\delta(\epsilon(\text{TabEstesa}, \text{Valori}, \text{TE}), 1) == \text{null}$ )
  then TE
  else Aggiungi (Tr_norm_dec(  $\delta(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, \text{TE}), 1), 1)$ ,
                              ESP,
                               $\tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, \text{TE}), 1), \text{ATT})$ ,
                               $\delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, \text{TE}), 1)$  ),
                Fnorm1_dec(ESP, ATT, RiduciTabella(TE)))
```

Tr_norm_dec (TP, ESP, POS, NA) is

```
NuovaTupla( TP,
             $\delta(\text{TP}, \text{POS}) / 10^{\text{ESP}}$  ,
            POS , null , NA)
```

Osservazione:

Si noti che in tutti i metodi di normalizzazione, lo schema di implementazione è lo stesso, e ricorrono sempre le tre funzioni seguenti:

- *Fnorm_metodo* : è la funzione di piú alto livello, definita per ricorsione sul numero di attributi da normalizzare
- *Fnorm1_metodo* : è la funzione di normalizzazione definita un singolo attributo, ricorsiva sul numero di tuple della tabella
- *Tr_norm_metodo* : implementa la vera e propria normalizzazione su una singola tupla

3.4.16 Il campionamento

3.4.16.1 La riduzione dei dati

In un ambiente di data mining la dimensione dei dati è una questione cruciale. I processi di estrazione di conoscenza possono infatti richiedere tempi inaccettabili, se applicati direttamente alle intere collezioni di dati, le quali raggiungono in genere notevoli dimensioni. La *riduzione dei dati* rappresenta allora un importante strumento nella fase di preparazione alle operazioni di data mining. Essa permette di ottenere una rappresentazione ridotta di un insieme di dati, che occupa un volume molto inferiore, mantenendo ciononostante una forte integrità rispetto i dati originali.

La **riduzione orizzontale** consiste nell'eliminazione di attributi, ossia di colonne di una tabella. A questo scopo è necessario individuare quali possono essere gli attributi irrilevanti ai fini dell'estrazione di una particolare istanza di modello di conoscenza.

Con la **riduzione verticale** invece si intende una trasformazione di una tabella che ne diminuisca il numero di tuple. Sono possibili diverse alternative, tra cui il

tradizionale raggruppamento rappresentato dall'operazione *Group by*, presente in tutti i linguaggi per basi di dati, associato al calcolo di valori aggregati. Una seconda possibilità, che sfrutta le funzionalità di data mining del nostro ambiente, consiste nel partizionare l'insieme di dati in clusters, e calcolare successivamente un qualche valore aggregato, raggruppando i dati per cluster.

Infine, una trattazione a parte è richiesta dalla riduzione verticale attraverso il *campionamento*.

3.4.16.2 Riduzione verticale: il campionamento

Nonostante la semplicità dell'operazione, il campionamento dei dati è una questione molto importante. Esso è spesso indispensabile per poter effettuare l'estrazione di modelli di conoscenza da collezioni di dati, che avrebbero altrimenti dimensioni intrattabili. Si noti inoltre che la strategia di campionamento può essere cruciale, perché influisce sui risultati che si ottengono nella successiva fase di data mining. Bisogna perciò sempre tenere presente quale è lo scopo di questa riduzione verticale, ossia quale è l'uso che si intende fare dei dati selezionati. È quindi in genere opportuno effettuare un nuovo campionamento sulla collezione di partenza, o quanto meno riconsiderarne la necessità, ogni volta che si intende effettuare una nuova ed indipendente sequenza di operazioni di data mining.

Data una collezione D di dati, contenente N elementi, alcuni metodi comuni di campionamento sono:

1. Il **campionamento casuale senza sostituzione** (**SRSWOR** – Simple Random Sample Without Replacement) **di dimensione n** : si costruisce un sottoinsieme di D che contiene n elementi fra gli N appartenenti a D ($n < N$), dove la probabilità di estrazione di ogni elemento in D è pari a $1/N$, ossia tutti gli elementi sono equiprobabili.
2. Il **campionamento casuale con sostituzione** (**SRSWR** – Simple Random Sample With Replacement) **di dimensione n** : è simile al campionamento SRSWOR, ma differisce solamente nel fatto che un elemento selezionato

non è “rimosso” dall’insieme di partenza D , e può essere selezionato anche successivamente, con la stessa probabilità.

3. Il **campionamento di clusters**: si prescinde dalla partizione degli elementi di D in un certo numero M di clusters, e viene applicato uno dei due metodi SRS alla collezione di clusters. Si ottiene così un sottoinsieme di m clusters completi di tutti i loro elementi componenti. In questo modo non vanno perse le correlazioni inter-cluster.

4. Il **campionamento stratificato**: anche questo metodo prescinde dalla specifica di un attributo, il quale partiziona i dati in diversi *strati*. Il campionamento stratificato consiste nell’applicazione di un metodo SRS ad ogni strato. Il campione risultante sarà perciò un sottoinsieme di D di rappresentanza per tutti i possibili strati.

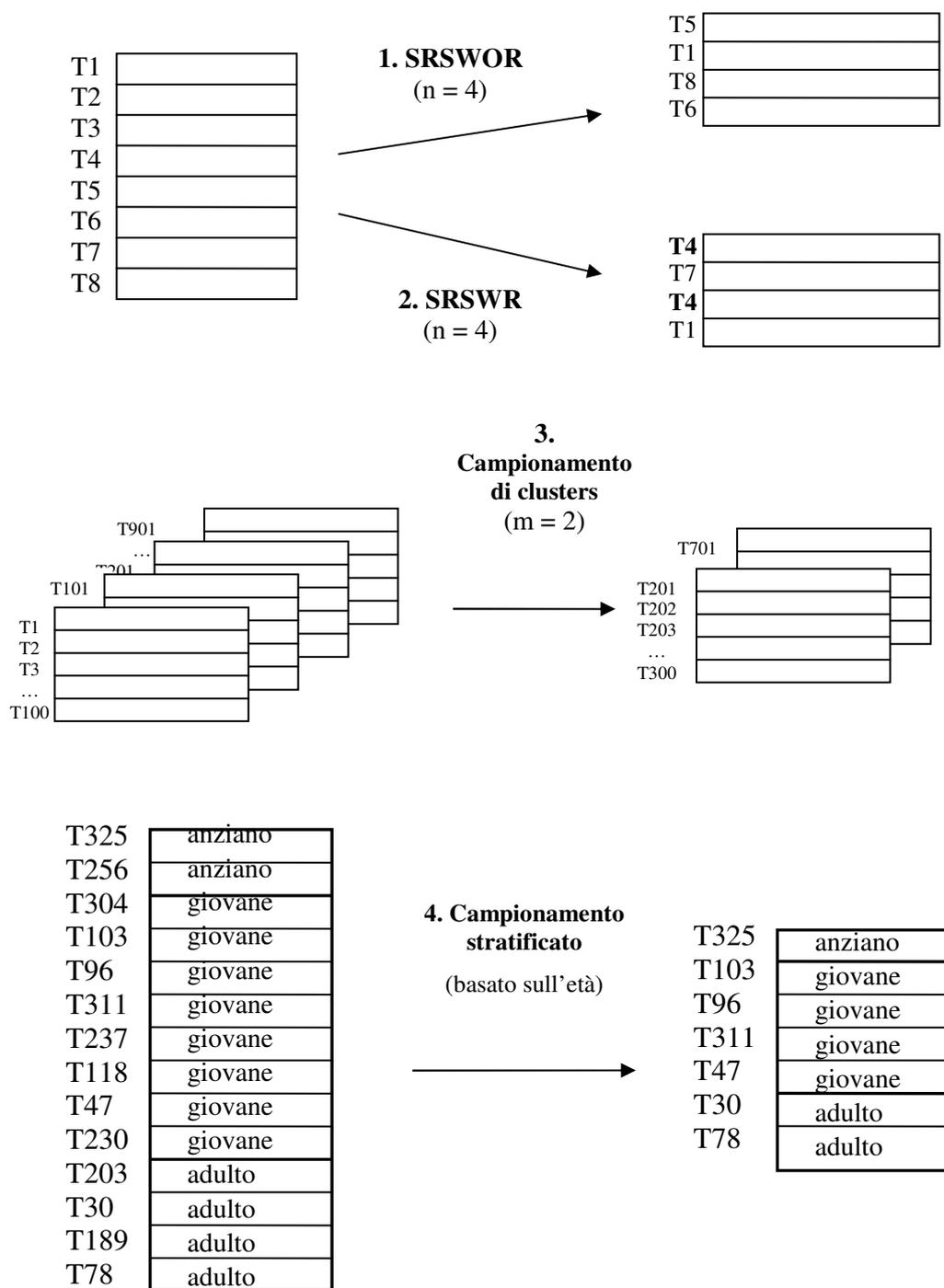


Figura 3.3: I metodi di campionamento

Sintassi :

```

<Campionamento> ::= SAMPLE <tabella_estesa>
                    <Metodo_campionamento>
                    USE <Algoritmo_campionamento>
<Metodo_campionamento> ::= SIMPLE
                            | BY CLUSTER <attributo>
                            | STRATIFIED ON <attributo>
<Algoritmo_campionamento> ::= (SRSWOR | SRSWR)
                                WITH (<n> ITEMS | INDEX <reale>)

```

Descrizione del costrutto :

Il costrutto permette di utilizzare tre diversi metodi di campionamento, specificati attraverso il tag *<Metodo_campionamento>*. La variante *SIMPLE* rappresenta il campionamento casuale delle tuple, con un algoritmo di tipo SRSWOR oppure SRSWR. È possibile fissare la cardinalità dell'insieme campione tramite una costante numerica (*WITH <n> ITEMS*), oppure tramite un indice di proporzionalità (*WITH INDEX <reale>*), inteso come il fattore di proporzione tra la cardinalità del campione e la cardinalità dell'insieme di partenza, ed è quindi compreso tra 0 e 1. Il numero di tuple selezionate corrisponde al minimo intero tale che l'indice di proporzionalità sia pienamente raggiunto.

La seconda e la terza variante presumono la partizione dell'insieme di partenza attraverso i valori di un certo attributo *Att*. Il metodo di campionamento corrispondente alla variante *BY CLUSTER <Att>* indica che, invece di considerare le singole tuple, l'algoritmo SRSWOR oppure SRSWR campiona un sottoinsieme dei clusters. Questa volta la costante numerica oppure l'indice di proporzionalità indicano il numero dei

clusters campione, piuttosto che quello delle tuple. Una volta selezionato un cluster, vengono estratte tutte le tuple appartenenti ad esso.

Infine la variante di campionamento stratificato, associata al tag *STRATIFIED ON <attributo>*, considera un cluster alla volta: se la soglia di campionamento è rappresentata da una costante numerica n , allora per ogni cluster vengono scelti n elementi, altrimenti se la soglia è data tramite un indice di proporzionalità, si termina il campionamento non appena si è raggiunto pienamente tale fattore.

Osservazioni:

Se la cardinalità n del campione dovesse essere superiore alla cardinalità dell'insieme origine, allora il comportamento è diverso a seconda dell'algoritmo applicato. Il campionamento SRSWOR produce un campione che corrisponde all'insieme di partenza, mentre il campionamento SRSWR si ferma solamente quando è stata raggiunta la soglia n , estraendo a caso, ad ogni passo, un elemento dell'insieme, anche se fosse già stato estratto in precedenza. Può dunque avere senso fissare una soglia di campionamento superiore alla cardinalità dell'insieme origine.

Sintassi astratta :

SAMPLE *Tabella* metodo USE algoritmo

≡

SAMPLE_m (*Tabella*, [*Attributo*], *Algoritmo*, *TipoSoglia*, *Soglia*)

dove

- m può essere uguale a SIMPLE, CLUSTER o STRATIFIED a seconda del metodo rappresentato dal tag in 'metodo';
- *Attributo* indica l'attributo di partizionamento nella seconda e terza variante;
- *Algoritmo* può essere uguale a SRSWOR oppure a SRSWR;

- *TipoSoglia* è uguale a INDEX oppure a ITEMS;
- *Soglia* rappresenta la costante numerica oppure il fattore di proporzione che rappresenta la cardinalità dell'insieme campione.

Implementazione :

1) Campionamento casuale semplice

SAMPLE_SIMPLE (Tab, Alg, TipoS, S) is

$$\gamma(\text{TabEstesa}, ((\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, \text{Tab}), 1), \\ \delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, \text{Tab}), 1), \\ \text{Campiona_tuple}(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, \text{Tab}), 1), \text{Alg}, \\ \text{TipoS}, S))))$$

Campiona_tuple effettua il campionamento dell'insieme rappresentato dalla sequenza di tuple *SeqValori*

Campiona_tuple(SeqValori, Alg, TipoSoglia, Soglia) = SeqCampione

dove

- *Alg* indica quale dei due algoritmi di campionamento descritti precedentemente viene applicato a *SeqValori*: SRSWOR oppure SRSWR;
- *SeqCampione* è una sottosequenza di *SeqValori*, che ha cardinalità limitata da una soglia:

se *TipoSoglia* == 'INDEX'

allora $\text{Cardinalità}(\text{SeqCampione}) = \text{Min}\{ k \mid$

$k/\text{Cardinalità}(\text{SeqValori}) \geq \text{Soglia}\}$

se *TipoSoglia* == 'ITEMS' e *Alg* == SRSWOR
 allora $\text{Cardinalità}(\text{SeqCampione}) = \text{Min}\{ \text{Cardinalità}(\text{SeqValori}), \text{Soglia} \}$

se *TipoSoglia* == 'ITEMS' e *Alg* == SRSWR
 allora $\text{Cardinalità}(\text{SeqCampione}) = \text{Soglia}$

2) Campionamento di clusters

SAMPLE_CLUSTER (Tab, Att, Alg, TipoS, S) is

$\gamma(\text{TabEstesa}, ((\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, \text{Tab}), 1),$
 $\delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, \text{Tab}), 1),$
 $\text{Campiona_clusters}(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, \text{Tab}), 1),$
 $\text{Att}, \text{Tab}, \text{Alg}, \text{TipoS}, \text{S})))$

Campiona_clusters effettua il campionamento dei clusters rappresentati dai valori dell'attributo *Att*, e per ogni cluster selezionato restituisce la sequenza delle tuple contenute in esso; il risultato è una sottosequenza di *SeqValori*

Campiona_clusters(SeqValori, Att, Tab, Alg, TipoSoglia, Soglia) is

$\sigma(\text{contenuto in } \text{Campiona_tuple}(\delta(\epsilon(\text{TabEstesa}, \text{Valori},$
 $\text{Valori_cluster}(\text{Tab}, \text{Att}), 1),$
 $\text{Alg}, \text{TipoSoglia}, \text{Soglia}),$
 $\tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, \text{Tab}), 1), \text{Att}),$
 $\text{SeqValori})$

Valori_cluster restituisce una tabella contenente tutti i valori di Attributo che compaiono in Tabella

Valori_cluster(Tabella, Attributo) =

```
SELECT Attributo
FROM Tabella
GROUP BY Attributo
```

3) Campionamento stratificato

SAMPLE_STRATIFIED(Tab, Att, Alg, TipoS, S) is

```
 $\gamma$ (TabEstesa, ((  $\delta$ ( $\epsilon$ (TabEstesa,Attributi,Tab),1),
 $\delta$ ( $\epsilon$ (TabEstesa,NumAttr,Tab),1),
Campione_stratificato(  $\delta$ ( $\epsilon$ (TabEstesa,Valori, Tab),1),
 $\tau$  ( $\delta$ ( $\epsilon$ (TabEstesa,Attributi,Tab),1),
Att),
Valori_cluster(Tab, Att)),1),
Alg, TipoS, S) )))
```

Campione_stratificato è una funzione ricorsiva che restituisce una sottosequenza di SeqValori:

per ogni strato, contenuto nella tabella TabStrati, effettua il campionamento delle tuple corrispondenti ad esso, e contenute in SeqValori; gli strati sono rappresentati dall'attributo in posizione PosA

Campione_stratificato(SeqValori, PosA, TabStrati, Alg, TipoS, S) is

```
if ( $\delta$ ( $\epsilon$ (TabEstesa,Valori,TabStrati),1) == null)
then null
else  $\alpha$  ( Campiona_tuple(  $\sigma$  ( ==  $\delta$ ( $\delta$ ( $\epsilon$ (TabEstesa,Valori, TabStrati),1),1),
PosA, SeqValori),
Alg, TipoS, S),
```

Campione_stratificato(SeqValori, PosA, RiduciTabella(TabStrati),
Alg, TipoS, S))

3.4.17 Ordinamento per frequenza

Questo costrutto permette di ordinare le tuple di una tabella in base alla frequenza con cui i valori di un attributo compaiono. L'utilità di questo riordinamento sta nell'ausilio che esso offre nella ricerca degli *outliers*, i quali ricorrono con una frequenza bassa. Il problema degli outliers è in realtà una questione difficile da affrontare, e che non può prescindere da un'attenta analisi dei dati. Il costrutto in questione costituisce solamente un'ulteriore strumento di analisi dei dati.

Alla tabella riordinata viene anche aggiunto un nuovo attributo "Frequenza " + *Attributo*, dove *Attributo* è l'attributo su cui è stata ordinata la tabella. Esso riporta, per ogni tupla, la frequenza con cui è stato riscontrato, all'interno dell'intera tabella, il valore dell'attributo in questione. Assieme all'attributo viene aggiunto anche il relativo campo di estensione, il quale può essere utilizzato tipicamente per marcare la tuple che non raggiungono una certa soglia di frequenza minima.

Sintassi :

**<Ordina_per_frequenza> ::= SORT <tabella_estesa> BY FREQUENCY
ON <attributo>**

Semantica dichiarativa :

SORT_FRQ(Tab, Att) = TE

dove

1. Schema di TE

Lo schema di TE è esteso con l'attributo "Frequenza attributo", dove attributo è quello a cui si applica l'ordinamento, e con il relativo campo di estensione :

$$Fschema_AggiungiColonna(Tab, "Frequenza "+Att)$$

2. Contenuto "estensionale" di TE

Data la sequenza $s1$ di tuple della tabella Tab:

$$\delta(\epsilon(TabEstesa, Valori, Tab), 1)$$

per ogni t appartenente a $s1$ siano

$i(t)$ la sua posizione nella sequenza $s1$: $Posizione(t, s1)$

$j(t)$ la sua posizione nella tabella TE: $Posizione(t, \delta(\epsilon(TabEstesa, Valori, TE), 1))$

$a(t)$ il valore dell'attributo Att di t : $\delta(t, \tau(\delta(\epsilon(TabEstesa, Attributi, TE), 1), Att))$

$f(t)$ il valore dell'attributo "Frequenza "+Att di t :

$$\delta(t, \tau(\delta(\epsilon(TabEstesa, Attributi, TE), 1), "Frequenza "+Att))$$

Allora per ogni t deve valere che

1. Deve esistere uno e un solo valore $j(t)$
2. $f(t) = Cardinalità(a(t), \rho(s1, (\tau(\delta(\epsilon(TabEstesa, Attributi, Tab), 1), Att))))$
3. per ogni coppia di tuple $t1$ e $t2$ appartenenti a $s1$
 - se $a(t1) = a(t2)$ allora $j(t1) < j(t2) \Leftrightarrow i(t1) < i(t2)$
 - se $a(t1) \neq a(t2)$ e $f(t1) < f(t2)$ allora $j(t1) < j(t2)$
 - se $a(t1) \neq a(t2)$ e $f(t1) > f(t2)$ allora $j(t1) > j(t2)$

- se $a(t1) \neq a(t2)$ e $f(t1) = f(t2)$ allora $j(t1) < j(t2) \Leftrightarrow i(t1) < i(t2)$ ⁶

□

3.4.18 I duplicati

Per affrontare il problema dei duplicati sono stati introdotti nel nostro linguaggio due costrutti di supporto. Essi verificano l'eventuale uguaglianza, rispetto ad un insieme di attributi, tra tuple adiacenti. Il primo costrutto permette di marcare i duplicati individuati in questo modo, mentre il secondo elimina dalla tabella, per ogni tupla, tutti i duplicati successivi. È importante notare che entrambi i costrutti prescindono da un ordinamento effettuato in precedenza, attraverso un tradizionale *order by* in stile SQL.

Anche se i duplicati sono considerati come tali in base ad una condizione di uguaglianza pura, ancora una volta si può sfruttare la potenzialità delle clausole di riscrittura per estendere la ricerca di duplicati con una condizione di “similitudine”. Basta infatti precedere i due costrutti in questione con delle clausole di riscrittura, mettendo in evidenza le componenti che si considerano essenziali per valutare l'eventuale duplicazione di un valore.

3.4.18.1 MARK DUPLICATES

Sintassi :

<Marca_duplicati> ::= MARK DUPLICATES

⁶ Il riordinamento è minimo: non altera l'ordine che c'è nella tabella di origine tra i dati aventi lo stesso valore e tra i dati che ricorrono con la stessa frequenza.

IN *<tabella_estesa>*
ON (*<SeqAttributi>*)
[WITH CODE *<n>***]**
<SeqAttributi> ::= *<attributo>* , *<SeqAttributi>* | *<attributo>*

Descrizione del costrutto :

La sequenza di attributi (*Att1, Att2, ...*) specificati attraverso il tag *ON* (*<SeqAttributi>*) indica gli attributi rispetto ai cui valori si verifica se le tuple adiacenti sono duplicate. Sono marcate tutte le tuple che hanno un duplicato adiacente, ossia nella tupla precedente oppure successiva. Se una tupla viene marcata, le informazioni di marcatura sono inserite in tutti gli attributi coinvolti nella valutazione di duplicato, ossia quelli contenuti nel tag *ON* (*<SeqAttributi>*).

Sintassi astratta :

MARK DUPLICATES IN *Tab* ON (*Att1,Att2,...*) [WITH CODE *Cod*]
 ≡ MARK_D (*Tab*, (*Att1,Att2,...*), *Marca*)

dove

- *Marca* == “NO_MARK” se non c’è il tag WITH CODE;
- *Marca* == *Cod* se i duplicati sono marcati attraverso il tag WITH CODE
Cod;

Semantica dichiarativa :

MARK_D(*Tab*, *SeqAtt*, *Marca*) = TE

dove TE è costruita nel seguente modo:

- sia *seqPos* la sequenza delle posizioni corrispondenti agli attributi in SeqAtt :

$$\mathit{seqPos} = (\mathit{Pos}_1, \mathit{Pos}_2, \mathit{Pos}_3, \dots)$$
dove $\mathit{Pos}_i = \tau(\delta(\epsilon(\mathit{TabEstesa}, \mathit{Attributi}, \mathit{Tab}), 1), \delta(\mathit{SeqAtt}, i))$
- sia *s* la sequenza di tuple della tabella Tab:

$$s = \delta(\epsilon(\mathit{TabEstesa}, \mathit{Valori}, \mathit{Tab}), 1)$$
- per ogni coppia di tuple, *t1* e *t2*, adiacenti nella sequenza *s*, se esse hanno gli stessi valori in corrispondenza delle posizioni contenute in *seqPos* :
se : $\rho(\mathit{t1}, \mathit{seqPos}) == \rho(\mathit{t2}, \mathit{seqPos})$

allora vengono marcati, in entrambe le tuple, tutti gli attributi specificati in SeqAtt :

allora : per ogni **p** appartenente a **seqPos**
 $\eta(\text{"mark_duplicates"} + \mathit{SeqAtt} + \delta(\mathit{t1}, \mathbf{p}) + [\text{"Codice : " + Cod}], \mathit{t1},$
 $\mathbf{p} + \delta(\epsilon(\mathit{TabEstesa}, \mathit{NumAttributi}, \mathit{Tab}), 1)$)
 $\eta(\text{"mark_duplicates"} + \mathit{SeqAtt} + \delta(\mathit{t2}, \mathbf{p}) + [\text{"Codice : " + Cod}], \mathit{t2},$
 $\mathbf{p} + \delta(\epsilon(\mathit{TabEstesa}, \mathit{NumAttributi}, \mathit{Tab}), 1)$)

□

3.4.18.2 MERGE DUPLICATES

Sintassi :

```
<Fondi_duplicati> ::= MERGE DUPLICATES
                        IN <tabella_estesa>
                        ON ( <SeqAttributi> )
                        [WITH CODE <n>]
<SeqAttributi> ::= <attributo> , <SeqAttributi> | <attributo>
```

Descrizione del costrutto :

Analogamente al costrutto di marcatura dei duplicati, la sequenza di attributi (*Att1*, *Att2*, ...), specificati attraverso il tag *ON* (*<SeqAttributi>*), indica gli attributi rispetto ai quali valori si verifica se le tuple adiacenti sono duplicate. Sono eliminate dalla tabella tutte le tuple che hanno un duplicato che lo precede, e rimane solamente la prima copia per ogni sottosequenza di duplicati adiacenti. Anche in questa trasformazione si presume che le tuple siano già ordinate rispetto agli attributi presi in considerazione nella verifica di duplicazione.

Se è specificato il tag *WITH CODE* *<n>* allora una sottosequenza di tuple, per essere considerata una sequenza di duplicati da “fondere”, deve anche essere marcata con il codice *n*. Si intende che una sottosequenza di tuple è marcata, se per ogni sua tupla sono marcati tutti gli attributi su cui si verifica l’eventuale duplicazione, ossia quelli specificati nel tag *ON* (*<SeqAttributi>*).

Sintassi astratta :

MERGE DUPLICATES IN *Tab* ON (*Att1,Att2,...*) [*WITH CODE Cod*]

≡ MERGE_D (*Tab*, (*Att1,Att2,...*), *Marca*)

dove

- *Marca* == *null* se non c’è il tag WITH CODE;
- *Marca* == *Cod* se è richiesto che i duplicati siano marcati con il codice *Cod*;

Semantica dichiarativa :

MERGE_D(*Tab*, *SeqAtt*, *Marca*) = TE

dove TE è costruita nel seguente modo:

- sia *seqPos* la sequenza delle posizioni corrispondenti agli attributi in SeqAtt :

$$\mathit{seqPos} = (\mathit{Pos}_1, \mathit{Pos}_2, \mathit{Pos}_3, \dots)$$

$$\text{dove } \mathit{Pos}_i = \tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, \text{Tab}), 1), \delta(\text{SeqAtt}, i))$$

- sia *s* la sequenza di tuple della tabella Tab:

$$s = \delta(\epsilon(\text{TabEstesa}, \text{Valori}, \text{Tab}), 1)$$

- per ogni coppia di tuple adiacenti, *t1* e *t2*, nella sequenza *s*, dove *t1* precede *t2*, se esse hanno gli stessi valori in corrispondenza delle posizioni contenute in *seqPos* :

$$\text{se: } \quad \rho(\mathbf{t1}, \mathit{seqPos}) == \rho(\mathbf{t2}, \mathit{seqPos})$$

allora si elimina da Tab la tupla *t2*, sia nel caso in cui Marca fosse uguale a 'null', sia nel caso in cui Marca avesse il valore 'cod' e *t1* e *t2* fossero marcate con tale codice.

allora :

$$\text{se: } \quad \text{Marca} == \text{'null'}$$

$$\text{allora: } \quad \lambda(s, \tau(s, \mathbf{t2}))$$

altrimenti :

$$\text{se: } \quad \text{per ogni } \mathbf{p} \text{ appartenente a } \mathit{seqPos}$$

“Codice : ”+ *Marca*

compare in

$$\delta(\mathbf{t1}, \mathbf{p} + \delta(\epsilon(\text{TabEstesa}, \text{NumAttributi}, \text{Tab}), 1))$$

ed anche in

$$\delta(\mathbf{t2}, \mathbf{p} + \delta(\epsilon(\text{TabEstesa}, \text{NumAttributi}, \text{Tab}), 1))$$

$$\text{allora: } \quad \lambda(s, \tau(s, \mathbf{t2}))$$

- si riprende dall'ultimo punto, fino a che non si trovano più duplicati adiacenti.

□

3.4.19 Operatori SQL

Naturalmente è possibile effettuare una qualunque interrogazione attraverso gli operatori SQL anche sulle *tabelle estese*. Il linguaggio di interrogazione SQL, ad un livello alto, è costituito da operazioni complesse, ma sintetiche, che sono frequenti nell'uso di basi di dati relazionali. Queste possono tuttavia essere ricondotte ad operazioni più semplici, dette *operatori primitivi*, e che formano l'*algebra relazionale*. Se l'insieme di operatori primitivi è minimo, l'algebra è detta minima. È possibile allora dare la semantica del linguaggio SQL rispetto al nuovo dominio, costituito dalle tabelle estese, semplicemente ridefinendo l'algebra relazionale sul nuovo dominio.

Si è deciso di adottare un approccio dichiarativo, piuttosto che uno operativo fondato sulle sequenze, per rendere più leggibile ed intuitiva la semantica SQL sulle tabelle estese, riconducendola ai tradizionali operatori SQL delle basi di dati relazionali.

3.4.19.1 Gli operatori primitivi

Di seguito verrà data la definizione di un insieme minimo di operatori primitivi. Essi si applicano a tabelle estese, e restituiscono come risultato ancora tabelle estese, formando perciò un'algebra minima.

Gli operatori primitivi sono: *ridenominazione*, *unione*, *differenza*, *proiezione*, *restrizione* e *prodotto*.

Nelle definizioni che seguono si useranno le seguenti notazioni:

- i simboli R, S ecc. sono nomi di tabelle estese; il simbolo $R[]$ denota la relazione corrispondente alla tabella estesa R , privata della sua estensione;
- A, A_1, A_2 ecc. sono nomi di attributi e X, Y sono insiemi di attributi;
- i simboli t, u ecc. denotano ennuple *estese*, ovvero righe delle tabelle estese;
- $t[]$ denota l'ennupla t privata della sua estensione, e $t[Est]$ denota la parte di estensione di t ;
- $t[A]$ denota il valore dell'attributo A nell'ennupla t , $t[Est A]$ denota l'estensione relativa all'attributo A nell'ennupla t , e $t[X]$ denota l'ennupla ottenuta da t considerando solo gli attributi in X .

Ridenominazione

Questo operatore si usa per cambiare il tipo di una tabella estesa R , modificando il nome di un attributo.

Siano X gli attributi di R , A e B due attributi tali che $A \in X$ e $B \text{ non } \in X$. R con A ridenominato B , denotata con $\delta_{A \rightarrow B}(R)$, è una tabella estesa con attributi $X - \{A\} - \{Est A\} \cup \{B\} \cup \{Est B\}$ definita come segue:

$$\delta_{A \rightarrow B}(R) = \{ t \mid \exists u \in R \text{ tale che } t[B] = u[A] \wedge t[Est B] = u[Est A] \wedge t[C] = u[C] \text{ se } C \neq B \text{ e } C \neq Est B \}.$$

Unione

$$R \cup S = \{ t \mid t \in R \vee t \in S \},$$

con R e S tabelle estese uniformi nello schema. Restituisce la tabella estesa ottenuta facendo l'unione delle ennuple estese di R con quelle di S .

Differenza

$$R - S = \{ t \mid t \in R \wedge t[] \text{ non } \in S[] \},$$

con R e S tabelle estese uniformi nello schema. Restituisce la tabella estesa contenente le ennuple di R non presenti in S . La differenza è valutata sulla parte priva di estensione, e nel risultato la parte di estensione è quella di R .

Proiezione

$$\pi_{A_1, A_2, \dots, A_m}(R) = \{ t[A_1, A_2, \dots, A_m, Est A_1, Est A_2, \dots, Est A_m] \mid t \in R \},$$

con A_1, A_2, \dots, A_m attributi di R , e $Est A_1, Est A_2, \dots, Est A_m$ le relative estensioni. Restituisce una tabella estesa i cui elementi sono la copia delle ennuple estese di R proiettate sugli attributi A_1, A_2, \dots, A_m , e sulle estensioni di tali attributi $Est A_1, Est A_2, \dots, Est A_m$.

Restrizione

$$\sigma_{\Phi}(R) = \{ t \mid t \in R \wedge \Phi(t) \}.$$

Restituisce una tabella estesa dello stesso tipo di R i cui elementi sono la copia delle ennuple estese di R che soddisfano la condizione. La condizione Φ è una formula proposizionale definita come segue:

- $A_i \theta A_j$, con A_i e A_j attributi di R e θ un operatore di confronto $\{<, >, =, \neq, \leq, \geq\}$;
- $A_i \theta c$, oppure $c \theta A_i$, con θ un operatore di confronto e c una costante in $dom(A_i)$;
- se φ e ψ sono formule, allora lo sono anche $\varphi \wedge \psi$, $\varphi \vee \psi$ e $\neg \psi$.

Osservazione:

Nell'algebra DPL sono anche previste le condizioni che coinvolgono la marcatura:

- *Exception*;
- *Exception* ε , con ε un codice di marcatura;
- *Exception* $[\varepsilon]$ on A_i ,

dove

- | | | |
|--|-------------------|---|
| a) $\psi(t) = \text{Exception}(t) = \mathbf{true}$ | \Leftrightarrow | esiste, in t , un attributo marcato; |
| b) $\psi(t) = \text{Exception } \varepsilon (t) = \mathbf{true}$ | \Leftrightarrow | esiste, in t , un attributo marcato con codice ε ; |
| c) $\psi(t) = \text{Exception } [\varepsilon] \text{ on } A_i (t) = \mathbf{true}$ | \Leftrightarrow | in t , l'attributo A_i è marcato [con codice ε]. |

Prodotto

$$R \times S = \{ t[]u[]t[Est]u[Est] \mid t \in R \wedge u \in S \},$$

con R e S tabelle estese con attributi distinti. Restituisce una tabella estesa con elementi ottenuti concatenando ogni ennupla di R , compresa la sua estensione, con tutte quelle di S , comprensive di estensione.

3.4.19.2 Operatori non derivati

Esistono tuttavia alcuni operatori SQL non definibili a partire dagli operatori primitivi dell'algebra relazionale, ma utili nelle applicazioni di basi di dati. Tra questi prenderemo in considerazione soltanto l'operatore **GROUP BY**, usato per esprimere funzioni di aggregazione su sottoinsiemi di ennuple, quali la media, il massimo, il minimo, il conteggio sul numero di elementi, la somma ed nel linguaggio DPL è prevista anche la concatenazione di stringhe.

Sul dominio tradizionale costituito dalle tabelle relazionali, una funzione di aggregazione è espressa attraverso la seguente espressione:

R groupBy X creating ($A_1 = exp_1, \dots, A_k = exp_k$),

dove X denota attributi della relazione R ed exp_i sono espressioni ottenute a partire dagli attributi in X e dall'applicazione di operatori su insiemi, quali la media, il massimo, ecc. Le funzioni di aggregazione sono valutate in questo modo:

- si partizionano le ennuple di una relazione R in un insieme di gruppi, mettendo nello stesso gruppo tutte le ennuple che coincidono su tutti gli attributi in X ;
- si calcolano le espressioni di aggregazione per ogni gruppo;
- si restituisce una relazione con un'ennupla per ogni gruppo.

Sul nuovo dominio costituito dalle tabelle estese, le funzioni di aggregazione sono valutate nello stesso modo, considerando solamente la parte priva di estensione. La tabella estesa che costituisce il risultato non conterrà alcuna informazione di estensione.

3.4.20 Restrizione WHERE

Sulle *trasformazioni locali* ha senso definire una restrizione del tipo *WHERE* in stile SQL. Per questa classe, l'elaborazione consiste nell'applicazione di una funzione *tr_f* ad ogni tupla della tabella in ingresso (vedi paragrafo 3.3.2); ogni tupla viene così trasformata in modo indipendente dalle altre. Si può allora controllare l'applicazione di tale funzione "locale" *tr_f* attraverso una condizione di restrizione sulla tupla. Tale condizione, come nella clausola *WHERE* nelle interrogazioni SQL, è rappresentata da un'espressione booleana formata da operatori di confronto applicati ad attributi e costanti, dalla composizione logica di espressioni booleane, ed anche in questo caso, è prevista la condizione di marcatura sulla tupla. La condizione di restrizione, rappresentata dalla clausola *WHERE* (*condizione*), va semplicemente aggiunta alla fine del costruito.

Dato lo schema di implementazione di una trasformazione locale *f*:

- a) $f(\text{tabella_vuota}) = \text{tabella_vuota}$
- b) $f(\text{TE}) = \text{Aggiungi}(\text{tr_f}(\text{prima_tupla_di_TE}), f(\text{TE-1}))$

si definisce la restrizione *WHERE* di *f* nel seguente modo:

$$f'(\Phi, \text{tabella_vuota}) = \text{tabella_vuota}$$

e

$$f'(\Phi, \text{TE}) = \text{Aggiungi}(\text{tr_f}'(\Phi, \text{prima_tupla_di_TE}), f'(\Phi, \text{TE-1}))$$

dove

- Φ è una condizione di *restrizione SQL* (vedi paragrafo 3.4.19.1)

Capitolo 4

Proprietà del linguaggio

Dopo avere definito la semantica operativa del linguaggio, nel presente capitolo verranno enunciate e dimostrate alcune proprietà algebriche interessanti. Il valore di tali proprietà consiste principalmente nell'evidenziare le caratteristiche strutturali di alcuni costrutti, le quali inducono delle possibili ottimizzazioni di implementazione.

La strategia di dimostrazione è strettamente legata allo stile di definizione della semantica, portando a scegliere, come strumenti, la tecnica dell'induzione e della dimostrazione per casi. Inoltre, alcune proprietà hanno valenza per diversi operatori, inducendo una classificazione dei costrutti. Per questo motivo si è anche preferito, in fase di definizione semantica, dare una implementazione strutturata al linguaggio, permettendo così di dimostrare, simultaneamente, alcune proprietà per una intera classe di costrutti. Allo stesso tempo, la dimostrazione ad un livello di astrazione superiore, permette anche una più intuitiva e profonda comprensione delle proprietà.

Sono di seguito dimostrate solo alcune proprietà, limitatamente al linguaggio DPL. Potrebbe invece rivelarsi interessante lo studio di correlazioni tra gli operatori di MQL, il linguaggio di interrogazione per l'estrazione di conoscenza all'interno del sistema KDDML, e gli operatori di DPL. Si noti che, essendo uniforme lo stile di definizione semantica dei due linguaggi, ed avendo adottato sempre la *sequenza* come modello di astrazione, è reso più agevole tale compito.

4.1 Alcune proprietà fondamentali

Di seguito vengono dimostrate alcune utili proprietà, le quali sono riferite alle operazioni derivate che ricorrono con una certa frequenza durante l'implementazione

del linguaggio (vedi paragrafo 3.3.1). Allo stesso modo in cui tali operazioni hanno costituito una base comune nell'implementazione di diversi costrutti, così le proprietà che seguono forniscono un supporto per la dimostrazione delle proprietà algebriche di diversi costrutti.

Le operazioni e le relazioni di ordinamento sul dominio delle tabelle estese sono quelle definite nel paragrafo 3.2.2.

Tesi :

Proprietà a)

Se T1 e T2 sono due tabelle estese, e T1 non è vuota, ossia se T1[Valori] ≠ null, allora vale che

$$\text{RiduciTabella}(T1 \cup_T T2) \equiv \text{RiduciTabella}(T1) \cup_T T2$$

□

Dimostrazione : (per induzione su T1)

Caso base: T1 possiede un solo valore v: T1[Valori] = (v)

RiduciTabella(T1 \cup_T T2)

≡ { per la def. di RiduciTabella }

$$\gamma(\text{TabEstesa}, ((\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, T1 \cup_T T2), 1), \\ \delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, T1 \cup_T T2), 1), \\ \lambda (\delta(\epsilon(\text{TabEstesa}, \text{Valori}, T1 \cup_T T2), 1), 1) \quad))))$$

≡ { per la def. di \cup_T e l'ipotesi: T1[Valori] = (v) }

$$\gamma(\text{TabEstesa}, ((\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, T2), 1), \\ \delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, T2), 1), \\ \delta(\epsilon(\text{TabEstesa}, \text{Valori}, T2), 1) \quad))))$$

≡ { per la def. di RiduciTabella }

T_2

≡ { per la def. di RiduciTabella e l'ipotesi: $T_1[\text{Valori}] = (v)$ e T_1 uniforme a T_2 nello schema }

$\text{RiduciTabella}(T_1) \cup_T T_2$

▪

Caso induttivo: $\#T_1[\text{Valori}] = n > 1$

Ipotesi induttiva : se $T_1' <_E T_1$ allora per ogni T_2' vale che

$\text{RiduciTabella}(T_1' \cup_T T_2') \equiv \text{RiduciTabella}(T_1') \cup_T T_2'$

$\text{RiduciTabella}(T_1 \cup_T T_2)$

≡ { siano T_1' e T_1'' t.c. $T_1' \cup_T T_1'' = T_1$ e $\#T_1''[\text{Valori}] > 0$ }

$\text{RiduciTabella}(T_1' \cup_T T_1'' \cup_T T_2)$

≡ { sia $T_2' = T_1'' \cup_T T_2$, per l'associatività dell'unione \cup_T }

$\text{RiduciTabella}(T_1' \cup_T T_2')$

≡ { $T_1' <_E T_1$ e applicando l'ipotesi induttiva su T_1' }

$\text{RiduciTabella}(T_1') \cup_T T_2'$

≡

$\text{RiduciTabella}(T_1') \cup_T T_1'' \cup_T T_2$

≡ { applicando ancora l'ipotesi induttiva su T_1' e per l'associatività dell'unione \cup_T }

$\text{RiduciTabella}(T_1) \cup_T T_2$

▪

Tesi :***Proprietà b)***

Se la tabella estesa T1 non è vuota, ossia se $\#T1[\text{Valori}] = n > 0$, allora esiste una tabella T0 di dimensione appena inferiore a T1, ossia con $\#T0[\text{Valori}] = n-1$, tale che $T0 <_r T1$, e *RiduciTabella(T1)* restituisce esattamente tale tabella T0.

□

Dimostrazione :

La tabella T0 è quella che si ottiene da T1 eliminando la prima tupla, ossia il primo elemento nella sequenza *Valori* di T1. *RiduciTabella*, per costruzione, restituisce esattamente tale tabella.

▪

Tesi :***Proprietà c)***

Date le due tabelle estese T1 e T2, vale che

$$\text{Aggiungi}(e, T1 \cup_r T2) \equiv \text{Aggiungi}(e, T1) \cup_r T2.$$

□

Dimostrazione : (per induzione su T1)

Caso base : $T1[\text{Valori}] = \text{null}$

Aggiungi(e, $T1 \cup_r T2$)

- \equiv { per la def. di \cup_T e $T1[\text{Valori}] = \text{null}$ }
 Aggiungi(e , $T2$)
- \equiv { per la def. di Aggiungi }
 $\gamma(\text{TabEstesa}, ((\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, T2), 1),$
 $\delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, T2), 1),$
 $\alpha((e), \delta(\epsilon(\text{TabEstesa}, \text{Valori}, T2), 1)))))$
- \equiv { sia T' la tabella che ha uno schema uniforme a $T2$, e contiene il solo valore e nella sequenza *Valori*, allora per la definizione di \cup_T all'inverso }
 $T' \cup_T T2$
- \equiv { per la def. di Aggiungi e l'ipotesi che $T1[\text{Valori}] = \text{null}$ }
 Aggiungi(e , $T1$) $\cup_T T2$

▪

Caso induttivo: $\#T1[\text{Valori}] = n > 0$

Ipotesi induttiva : se $T1' <_E T1$ allora per ogni $T2'$ vale che

Aggiungi(e , $T1' \cup_T T2'$) \equiv Aggiungi(e , $T1'$) $\cup_T T2'$

Aggiungi(e , $T1 \cup_T T2$)

\equiv { siano $T1'$ e $T1''$ t.c. $T1' \cup_T T1'' = T1$ e $\#T1''[\text{Valori}] > 0$ }

Aggiungi(e , $T1' \cup_T T1'' \cup_T T2$)

\equiv { sia $T2' = T1'' \cup_T T2$, per l'associatività dell'unione \cup_T }

Aggiungi(e , $T1' \cup_T T2'$)

$$\begin{aligned} &\equiv \{ T1' \prec_E T1 \text{ e applicando l'ipotesi induttiva su } T1' \} \\ &\quad \text{Aggiungi}(e, T1') \cup_T T2' \\ &\equiv \\ &\quad \text{Aggiungi}(e, T1') \cup_T T1'' \cup_T T2 \\ &\equiv \{ \text{applicando ancora l'ipotesi induttiva su } T1' \text{ e per l'associatività dell'unione } \cup_T \} \\ &\quad \text{Aggiungi}(e, T1) \cup_T T2 \end{aligned}$$

■

4.2 Le trasformazioni locali

Come già anticipato, si è riconosciuta l'importanza di un metodo strutturato nella definizione formale del linguaggio. In previsione di proprietà comuni a più operatori del linguaggio, si è introdotta nel paragrafo 3.3.2 la classe delle *trasformazioni locali*. Ricordiamo che la caratteristica fondamentale di questa classe di costrutti è che l'elaborazione di una tabella può essere considerata come una sequenza di elaborazioni indipendenti di singole tuple. Se indichiamo con tr_f la trasformazione di una tupla, in corrispondenza della trasformazione locale f applicata alla tabella estesa TE , allora l'implementazione di f segue lo schema seguente:

$$f(TE) = \text{Aggiungi}(tr_f(\text{prima_tupla_di_TE}), f(TE-1))$$

e

$$f(\text{tabella_vuota}) = \text{tabella_vuota} .$$

dove $TE-1$ indica la tabella TE privata della prima tupla. *Aggiungi* è una funzione che restituisce la tabella corrispondente al secondo parametro, in cui è stata inserita la tupla corrispondente al primo parametro. Sia la funzione *Aggiungi* che la funzione

RiduciTabella, corrispondente alla notazione *TE-1*, sono definite, assieme a tutte le operazioni derivate frequenti, nel paragrafo 3.3.1.

È evidente la natura ricorsiva di tale implementazione, che consente di dimostrare le proprietà per induzione sulla “dimensione” delle tabelle. Si noti anche come l’implementazione di questa classe di costrutti si presti ad una parallelizzazione secondo lo schema *pipeline*.

Prima di passare alla dimostrazione vera e propria, ricordiamo la definizione delle due operazioni sul dominio delle trasformazioni locali. Siano *f* e *g* due trasformazioni locali, allora sono definiti

il prodotto :

$$f \circ g (TE) = g (f (TE))$$

come la composizione delle due funzioni, e

la somma :

$$f + g (TE) = \text{Aggiungi} (\text{tr}_g (\text{tr}_f(\text{prima_tupla_di_TE})), \quad f+g(TE-1))$$

e

$$f + g (\text{tabella_vuota}) = \text{tabella_vuota}$$

come l’annidamento delle trasformazioni sulle tuple.

4.2.1 Prop. 1: Equivalenza tra sequenzializzazione ed annidamento

Tesi :

Proprietà 1

Per le trasformazioni locali, *f* e *g*, vale che

$$f \circ g (TE) \equiv f + g (TE) .$$

□

L'utilità di questa proprietà consiste nel poter scandire una sola volta la tabella, che generalmente è di notevoli dimensioni, anche quando si susseguono una serie di trasformazioni locali su di essa.

Dimostrazione :

Si procede per induzione sul dominio ben fondato delle tabelle estese, sul quale è definita la relazione di ordinamento parziale \leq_I , introdotta nel paragrafo 3.2.2.

Caso base: $TE == tabella_vuota$

$$\begin{aligned} f \circ g (TE) &\equiv g(f (TE)) \\ &\equiv \{ \text{def. } f \text{ locale e ipotesi} \} g(\text{tabella_vuota}) \\ &\equiv \{ \text{def. } g \text{ locale e ipotesi} \} \text{tabella_vuota} \\ &\equiv \{ \text{def. } f+g \text{ e ipotesi che } TE = \text{tabella_vuota} \} f + g (TE) \end{aligned}$$

▪

Caso induttivo: $\# TE[\text{Valori}] \neq 0$

Sia $TE' = TE - 1$; allora $TE' \leq_I TE$

Ipotesi induttiva: $f \circ g (TE') \equiv f + g (TE')$

$$\begin{aligned} &f + g (TE) \\ &\equiv \{ \text{def. } f+g \} \end{aligned}$$

Aggiungi (tr_g (tr_f(prima_tupla_di_TE)), f+g(TE-1))

≡ { ipotesi induttiva }

Aggiungi (tr_g (tr_f(prima_tupla_di_TE)), g(f(TE')))

≡ { def. g all'inverso e sia TE'' tale che

$prima_tupla_di_TE'' = tr_f(prima_tupla_di_TE)$ e

$TE''^{-1} = f(TE')$ }

g(TE'')

Non ci resta che verificare che TE'' è effettivamente $f(TE)$.

f(TE)

≡

Aggiungi(tr_f(prima_tupla_di_TE), f(TE-1))

≡ { def. di TE'' e di TE' }

Aggiungi(prima_tupla_di_TE'', TE''-1)

≡ { def. di Aggiungi }

TE''

▪

4.2.2 Prop. 2: Distributività delle trasformazioni locali rispetto all'unione

Tesi:

Proprietà 2

Data una trasformazione locale, e due tabelle estese $TE1$ e $TE2$ uniformi nello schema, vale che

$$f(TE1 \cup_T TE2) \equiv f(TE1) \cup_T f(TE2) .$$

□

Questa proprietà mette in evidenza come sia possibile applicare, nell'implementazione di trasformazioni locali, lo schema di parallelizzazione *farm*, oltre che quello *pipeline*, indotto dalla definizione.

Dimostrazione :

Daremo una dimostrazione per induzione sulla tabella $TE1$.

Caso base: $TE1[Valori] = \text{null}$ (la sequenza *Valori* di $TE1$ è vuota)

$$f(TE1 \cup_T TE2)$$

≡

$$\text{Aggiungi}(\text{tr_f}(\text{prima_tupla_di_}(TE1 \cup_T TE2)), f((TE1 \cup_T TE2)-1))$$

≡ {ipotesi : $TE1[Valori] = \text{null}$ e def. \cup_T }

$$\text{Aggiungi}(\text{tr_f}(\text{prima_tupla_di_}TE2), f(TE2-1))$$

≡ { def. di f }

$$f(TE2)$$

≡ {indichiamo con null_T la tabella estesa che ha lo stesso schema di $TE2$, ma è vuota }

$$(\text{null}_T) \cup_T f(TE2)$$

$\equiv \{ \text{per la def. di } f \text{ su tabella_vuota e per l'ipotesi: } TE1[\text{Valori}] = \text{null} \}$
 $f(TE1) \cup_T f(TE2)$

▪

Caso induttivo: $\#TE1[\text{Valori}] = n > 0$

Per la *proprietà b)* esiste una tabella $TE0$ t.c.

$\#TE0[\text{Valori}] = n-1$ e $TE0 <_l TE1$ per la quale vale la

ipotesi induttiva : $f(TE0 \cup_T TE2) \equiv f(TE0) \cup_T f(TE2)$

$f(TE1 \cup_T TE2)$

\equiv

Aggiungi($\text{tr_f(prima_tupla_di_}(TE1 \cup_T TE2))$, $f((TE1 \cup_T TE2)-1)$)

$\equiv \{ \text{sotto l'ipotesi : } TE1[\text{Valori}] \neq \text{null} \text{ vale che}$
 $\text{prima_tupla_di_}(TE1 \cup_T TE2) \equiv \text{prima_tupla_di_}TE1$,
 inoltre $(TE1 \cup_T TE2)-1 \equiv \text{RiduciTabella}(TE1 \cup_T TE2)$, a cui
 si applica la **proprietà a)** }

Aggiungi($\text{tr_f(prima_tupla_di_}TE1)$, $f(\text{RiduciTabella}(TE1) \cup_T TE2)$)

$\equiv \{ \text{per la } \mathbf{proprietà b)} \text{ , si può ora applicare l'ipotesi induttiva su } TE0 =$
 $\text{RiduciTabella}(TE1) \}$

Aggiungi($\text{tr_f(prima_tupla_di_}TE1)$, $f(\text{RiduciTabella}(TE1)) \cup_T f(TE2)$)

$\equiv \{ \text{per la } \mathbf{proprietà c)} \}$

Aggiungi($\text{tr_f(prima_tupla_di_}TE1)$, $f(\text{RiduciTabella}(TE1))$) $\cup_T f(TE2)$

\equiv

$$f(\text{TE1}) \cup_{\tau} f(\text{TE2})$$

▪

4.3 Le trasformazioni locali di schema

Nel paragrafo 3.4.5 sono state descritte le trasformazioni di schema, ed in particolare un insieme, formato dai seguenti costrutti: DIVIDE, SPLIT, MERGE, NEWATTRIBUTE REMOVE e RENAME. Tali trasformazioni, dette *locali di schema*, ereditano, oltre allo schema logico di implementazione, anche le proprietà delle trasformazioni locali.

Ricordiamo brevemente tale schema:

$$\begin{aligned} h(\text{TE}) &= \text{Funzione_locale}(\text{Modifica_di_schema}(\text{TE}, \text{parametri}), \text{parametri}) \\ &= h_s \circ h_l(\text{TE}) \end{aligned}$$

dove *Modifica_di_schema* (indicata anche con h_s) opera sugli elementi di schema di un oggetto di tipo *TabEstesa*, ossia su *Attributi* e *NumAttributi*. *F_locale* (indicata anche con h_l) è una trasformazione locale tradizionale, la quale opera sull'elemento *Valori*, procedendo tupla per tupla.

Inoltre, se f è una trasformazione locale elementare, allora l'operazione di somma, indotta (quindi compatibile) dalla somma sulle trasformazioni locali è definita nel modo seguente:

$$h + f(\text{TE}) = (h_s \circ h_l) + f(\text{TE}) = h_s \circ (h_l + f)(\text{TE}) .$$

4.3.1 Prop. 3: Equivalenza tra composizione ed annidamento

Tesi:

Proprietà 3

Vale il seguente corollario della **proprietà 1**:

sia f una trasformazione locale di schema e g una trasformazione locale elementare;
allora

$$f \circ g (TE) \equiv f + g (TE) .$$

□

Dimostrazione :

$$\begin{aligned} & f \circ g (TE) \\ \equiv & \{ \text{definizione dello schema logico di implementazione di } f \\ & \text{e associatività della composizione funzionale} \} \\ & f_s \circ (f_l \circ g) (TE) \\ \equiv & \{ \text{proprietà 1. applicata a } f_l \circ g ; f_l \text{ è una trasf. locale elementare!} \} \\ & f_s \circ (f_l + g) (TE) \\ \equiv & \{ \text{def. di somma indotta} \} \\ & f + g (TE) \end{aligned}$$

□

4.3.2 Prop. 4: Distributività rispetto all'unione**Tesi :****Proprietà 4**

Vale inoltre il corollario della **proprietà 2**:

se f è una trasformazione locale di schema, e $TE1$ e $TE2$ due tabelle estese uniformi nello schema, allora

$$f(TE1 \cup_T TE2) \equiv f(TE1) \cup_T f(TE2) .$$

□

Dimostrazione :

$$\begin{aligned}
& f(\text{TE1} \cup_{\text{T}} \text{TE2}) \\
\equiv & \\
& f_s \circ f_l (\text{TE1} \cup_{\text{T}} \text{TE2}) \\
\equiv & \\
& f_l (f_s (\text{TE1} \cup_{\text{T}} \text{TE2})) \\
\equiv & \{ \text{sia TE1}' \text{ la tabella TE1 modificata solamente nello schema, ossia TE1}' = f_s (\text{TE1}) \\
& \text{e sia TE2}' = f_s (\text{TE2}); \text{ si noti che TE1}' \text{ e TE2}' \text{ sono ancora uniformi nello} \\
& \text{schema!} \} \\
& f_l (\text{TE1}' \cup_{\text{T}} \text{TE2}') \\
\equiv & \{ \text{proprietà 2. applicata a } f_l \} \\
& f_l (\text{TE1}') \cup_{\text{T}} f_l (\text{TE2}') \\
\equiv & \{ \text{def. di TE1}' \text{ e TE2}' \} \\
& f(\text{TE1}) \cup_{\text{T}} f(\text{TE2})
\end{aligned}$$

▪

4.4 Proprietà della normalizzazione

Verranno di seguito dimostrate due proprietà che valgono per il costrutto corrispondente al metodo di *normalizzazione minimo-massimo*. È comunque possibile dimostrare le stesse proprietà anche per le altre varianti di normalizzazione.

4.4.1 Prop. 5: Commutatività**Tesi :**

Proprietà 5

Se $Att1 \neq Att2$, allora

$$\begin{aligned} & \text{NORM_MIN_MAX}(\text{min}, \text{max}, (\text{Att1}), \\ & \qquad \qquad \qquad \text{NORM_MIN_MAX}(\text{min}, \text{max}, (\text{Att2}), \text{Te})) \\ & \qquad \qquad \qquad \equiv \\ & \text{NORM_MIN_MAX}(\text{min}, \text{max}, (\text{Att2}), \\ & \qquad \qquad \qquad \text{NORM_MIN_MAX}(\text{min}, \text{max}, (\text{Att1}), \text{Te})) \end{aligned}$$

□

Dimostrazione :

Per comodità dimostriamo separatamente che

Tesi (Proprietà p):

$$\begin{aligned} & \text{Tr_norm_MM}(\\ & \quad \text{Tr_norm_MM}(\text{TP}, \text{MIN_ATT1}, \text{MAX_ATT1}, \text{MIN}, \text{MAX}, \text{POS1}, \text{NA}, \text{null}) , \\ & \quad \text{MIN_ATT2}, \text{MAX_ATT2}, \text{MIN}, \text{MAX}, \text{POS2}, \text{NA}, \text{null}) \\ & \qquad \qquad \qquad \equiv \\ & \text{Tr_norm_MM}(\\ & \quad \text{Tr_norm_MM}(\text{TP}, \text{MIN_ATT2}, \text{MAX_ATT2}, \text{MIN}, \text{MAX}, \text{POS2}, \text{NA}, \text{null}) , \\ & \quad \text{MIN_ATT1}, \text{MAX_ATT1}, \text{MIN}, \text{MAX}, \text{POS1}, \text{NA}, \text{null}) \end{aligned}$$

Dimostrazione: (prop. p)

$$\begin{aligned} & \text{Tr_norm_MM}(\\ & \quad \text{Tr_norm_MM}(\text{TP}, \text{MIN_ATT1}, \text{MAX_ATT1}, \text{MIN}, \text{MAX}, \text{POS1}, \text{NA}, \text{null}) , \\ & \quad \text{MIN_ATT2}, \text{MAX_ATT2}, \text{MIN}, \text{MAX}, \text{POS2}, \text{NA}, \text{null}) \\ & \equiv \{ \text{def. Tr_norm_MM} \} \end{aligned}$$

Tr_norm_MM(

NuovaTupla(TP,

$(\delta(TP, POS1) - MIN_ATT1 / MAX_ATT1 - MIN_ATT1) *$

$(MAX - MIN) + MIN,$

POS1 , null , NA),

MIN_ATT2, MAX_ATT2, MIN, MAX, POS2, NA, null)

$\equiv \{$ indichiamo con TP1 la tupla prodotta dalla funzione *NuovaTupla*, ossia $\varphi(NuovoValore1, TP, POS1)$, dove $NuovoValore1 = (\delta(TP, POS1) - MIN_ATT1 / MAX_ATT1 - MIN_ATT1) * (MAX - MIN) + MIN \}$

Tr_norm_MM(TP1 , MIN_ATT2, MAX_ATT2, MIN, MAX, POS2, NA, null)

$\equiv \{$ def. Tr_norm_MM $\}$

NuovaTupla(TP1,

$(\delta(TP1, POS2) - MIN_ATT2 / MAX_ATT2 - MIN_ATT2) * (MAX - MIN) + MIN,$

POS2 , null , NA)

$\equiv \{$ indichiamo con TP12 la tupla prodotta dalla funzione *NuovaTupla*, ossia $\varphi(NuovoValore2, TP1, POS2)$, dove $NuovoValore2 = (\delta(TP1, POS2) - MIN_ATT2 / MAX_ATT2 - MIN_ATT2) * (MAX - MIN) + MIN \}$

TP12

\equiv

$\varphi(NuovoValore2, \varphi(NuovoValore1, TP, POS1), POS2)$

$\equiv \{$ per la commutatività della sostituzione $\varphi \}$

$\varphi(NuovoValore1, \varphi(NuovoValore2, TP, POS2), POS1)$

\equiv

Tr_norm_MM(

Tr_norm_MM (TP, MIN_ATT2, MAX_ATT2, MIN, MAX, POS2, NA, null) ,

MIN_ATT1, MAX_ATT1, MIN, MAX, POS1, NA, null)

▪

La **dimostrazione della proprietà 5** è per induzione sulle tabelle.

Caso base: $Te[Valori] = \text{null}$

Dal momento che, per ogni valore di min , max , $Att0$, vale che:

$$\begin{aligned}
 & NORM_MIN_MAX(min, max, (Att0), Te) \\
 \equiv & \\
 & Fnorm1_MM(TrovaMinAttuale(Te, Att0), \\
 & \quad TrovaMaxAttuale(Te, Att0), \\
 & \quad min, max, Att0, \\
 & \quad Fnorm_MM(min, max, null, Te) , null) \\
 \equiv & \{ Fnorm_MM(min, max, null, Te) = Te \text{ e ipotesi del caso base: } Te[Valori] = \text{null} \} \\
 & Te \\
 & \cdot
 \end{aligned}$$

allora

$$\begin{aligned}
 & NORM_MIN_MAX(min, max, (Att1), NORM_MIN_MAX(min, max,(Att2), Te)) \\
 \equiv & NORM_MIN_MAX(min, max, (Att1), Te) \\
 \equiv & Te \\
 \equiv & NORM_MIN_MAX(min, max, (Att2), NORM_MIN_MAX(min, max,(Att1), Te))
 \end{aligned}$$

▪

Caso induttivo: $\# Te[Valori] \neq 0$

a) Trattiamo prima il caso in cui per ogni $Te0$ e $Att0$

$Trova[Min/Max]Attuale(Te0, Att0) \neq \text{null}$

$$NORM_MIN_MAX(min, max, (Att1), NORM_MIN_MAX(min, max,(Att2), Te)$$

$\equiv \{ \text{sia } Te2 \text{ la tabella } Te \text{ normalizzata rispetto ad } Att2: \text{ NORM_MIN_MAX}(\min, \max, (Att2), Te) \}$
 $\text{Fnorm1_MM}(\text{TrovaMinAttuale}(Te2, Att1),$
 $\text{TrovaMaxAttuale}(Te2, Att1),$
 $\min, \max, Att1,$
 $\text{Fnorm_MM}(\min, \max, \text{null}, Te2), \text{null})$
 $\equiv \{ \text{Fnorm_MM}(\min, \max, \text{null}, Te2) = Te2 \text{ e ipotesi } a) \text{ e sia}$
 $m1 = \text{TrovaMinAttuale}(Te2, Att1) \text{ e } M1 = \text{TrovaMaxAttuale}(Te2, Att1) \}$
 $\text{Aggiungi} (\text{Tr_norm_MM}(\delta(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, Te2), 1), 1),$
 $m1, M1, \min, \max,$
 $\tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, Te2), 1), Att1),$
 $\delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, Te2), 1),$
 $\text{null}),$
 $\text{Fnorm1_MM}(m1, M1, \min, \max, Att1, \text{RiduciTabella}(Te2), \text{null}))$
 $\equiv \{$
 $\text{Posti } m2 = \text{TrovaMinAttuale}(Te, Att2) \text{ e } M2 = \text{TrovaMaxAttuale}(Te, Att2) \text{ e } pos2$
 $= \tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, Te), 1), Att2) \text{ e } na = \delta(\epsilon(\text{TabEstesa}, \text{NumAttr}, Te), 1),$
 allora vale che
 $\delta(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, Te2), 1), 1)$
 $\equiv \text{Tr_norm_MM}(\delta(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, Te), 1), 1), m2, M2, \min, \max, pos2,$
 $na, \text{null})$
 $\}$
 $\text{Aggiungi} (\text{Tr_norm_MM}(\text{Tr_norm_MM}(\delta(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, Te), 1), 1), m2,$
 $M2, \min,$
 $\max, pos2, na, \text{null}),$
 $m1, M1, \min, \max,$
 $\tau(\delta(\epsilon(\text{TabEstesa}, \text{Attributi}, Te2), 1), Att1),$
 $na,$
 $\text{null}),$
 $\text{Fnorm1_MM}(m1, M1, \min, \max, Att1, \text{RiduciTabella}(Te2), \text{null}))$
 $\equiv \{$
 $\text{per la proprietà } p) \text{ e poiché}$

$$\begin{aligned} & \text{Fnorm1_MM}(m1, M1, \text{min}, \text{max}, \text{Att1}, \text{RiduciTabella}(\text{Te2}), \text{null}) \\ & \equiv \text{RiduciTabella}(\text{Fnorm1_MM}(m1, M1, \text{min}, \text{max}, \text{Att1}, \text{Te2}, \text{null})) \\ & \} \\ & \text{Aggiungi}(\text{Tr_norm_MM}(\text{Tr_norm_MM}(\delta(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, \text{Te}), 1), 1), m1, \\ & M1, \text{min}, \\ & \qquad \qquad \qquad \text{max}, \text{pos1}, \text{na}, \text{null}), \\ & \qquad \qquad \qquad m2, M2, \text{min}, \text{max}, \text{pos2}, \text{na}, \text{null}), \\ & \text{RiduciTabella}(\text{Fnorm1_MM}(m1, M1, \text{min}, \text{max}, \text{Att1}, \text{Te2}, \text{null})) \\ & \equiv \{ \\ & \quad \text{Posta } T' = \text{RiduciTabella}(\text{Te}) \text{ e} \\ & \quad \text{posta } T'' = \text{RiduciTabella}(\text{Fnorm1_MM}(m1, M1, \text{min}, \text{max}, \text{Att1}, \text{Te2}, \text{null}), \\ & \quad \text{allora vale che} \\ & \quad T'' \equiv \text{Fnorm1_MM}(m1, M1, \text{min}, \text{max}, \text{Att1}, \text{Fnorm1_MM}(m2, M2, \text{min}, \text{max}, \\ & \quad \text{Att2}, T', \text{null}), \text{null}) \equiv \text{NORM_MIN_MAX}(\text{min}, \text{max}, (\text{Att1}), \\ & \quad \text{NORM_MIN_MAX}(\text{min}, \text{max}, (\text{Att2}), T') \text{)} \\ & \quad \text{Si può ora applicare l'ipotesi induttiva su } T', \text{ essendo } T' < \text{Te}, \\ & \} \\ & \text{Aggiungi}(\text{Tr_norm_MM}(\text{Tr_norm_MM}(\delta(\delta(\epsilon(\text{TabEstesa}, \text{Valori}, \text{Te}), 1), 1), \\ & \qquad \qquad \qquad m1, M1, \text{min}, \text{max}, \text{pos1}, \text{na}, \text{null}), \\ & \qquad \qquad \qquad m2, M2, \text{min}, \text{max}, \text{pos2}, \text{na}, \text{null}), \\ & \qquad \qquad \qquad \text{Fnorm1_MM}(m2, M2, \text{min}, \text{max}, \text{Att2}, \text{RiduciTabella}(\text{Te1}), \text{null})) \\ & \equiv \{ \text{dove con Te1 si è indicata la tabella Te normalizzata rispetto ad Att1} \} \\ & \quad \text{NORM_MIN_MAX}(\text{min}, \text{max}, (\text{Att2}), \text{NORM_MIN_MAX}(\text{min}, \text{max}, (\text{Att1}), \text{Te})) \\ & \quad \blacksquare \end{aligned}$$

b) Consideriamo ora il caso in cui per qualche Te0 e Att0

$\text{Trova}[\text{Min/Max}]\text{Attuale}(\text{Te0}, \text{Att0}) = \text{null};$

sarà sufficiente trattare il caso $\text{TrovaMinAttuale}(\text{Te2}, \text{Att1}) = \text{null}$, dove Te2 è la tabella Te normalizzata rispetto ad Att2 . Le altre tre alternative e le possibili combinazioni si dimostrano in modo analogo.

$$\begin{aligned}
 & \text{NORM_MIN_MAX}(\text{min}, \text{max}, (\text{Att1}), \text{NORM_MIN_MAX}(\text{min}, \text{max}, (\text{Att2}), \text{Te})) \\
 \equiv & \{ \text{sia } \text{Te2} \text{ la tabella } \text{Te} \text{ normalizzata rispetto ad } \text{Att2}: \text{NORM_MIN_MAX}(\text{min}, \\
 & \quad \text{max}, (\text{Att2}), \text{Te}) \} \\
 & \text{Fnorm1_MM}(\text{TrovaMinAttuale}(\text{Te2}, \text{Att1}), \\
 & \quad \text{TrovaMaxAttuale}(\text{Te2}, \text{Att1}), \\
 & \quad \text{min}, \text{max}, \text{Att1}, \\
 & \quad \text{Fnorm_MM}(\text{min}, \text{max}, \text{null}, \text{Te2}), \text{null}) \\
 \equiv & \{ \text{Fnorm_MM}(\text{min}, \text{max}, \text{null}, \text{Te2}) = \text{Te2} \text{ e ipotesi } b): \\
 & \quad \text{TrovaMinAttuale}(\text{Te2}, \text{Att1}) = \text{null} \} \\
 & \text{Te2} \\
 \equiv & \{ \text{def. di NORM_MIN_MAX} \} \\
 & \text{Fnorm1_MM}(\text{TrovaMinAttuale}(\text{Te}, \text{Att2}), \\
 & \quad \text{TrovaMaxAttuale}(\text{Te}, \text{Att2}), \\
 & \quad \text{min}, \text{max}, \text{Att2}, \\
 & \quad \text{Fnorm_MM}(\text{min}, \text{max}, \text{null}, \text{Te}), \text{null}) \\
 \equiv & \{ \text{anche } \text{TrovaMinAttuale}(\text{Te}, \text{Att1}) = \text{null} \text{ e } \text{Te} = \text{NORM_MIN_MAX}(\text{min}, \\
 & \quad \text{max}, (\text{Att1}), \text{Te}) = \text{Te1} \} \\
 & \text{NORM_MIN_MAX}(\text{min}, \text{max}, (\text{Att2}), \text{NORM_MIN_MAX}(\text{min}, \text{max}, (\text{Att1}), \text{Te}))
 \end{aligned}$$

▪

Osservazione:

La proprietà di commutatività vale anche per le restanti quattro varianti di normalizzazione, e la dimostrazione segue lo stesso schema. C'è comunque da notare una limitazione nella variante di normalizzazione locale NORM_MIN_MAX_L , quando venga richiesta la marcatura con il tag *Exception*. In tal caso l'equivalenza

vale solo per la tabella non estesa, mentre i metadati contenuti nell'estensione potrebbero non essere equivalenti.

4.4.2 Prop. 6: Equivalenza tra sequenzializzazione ed annidamento

Tesi :

Proprietà 6

$$\begin{aligned}
 & \text{NORM_MIN_MAX}(\text{min}, \text{max}, (\text{Att1}), \\
 & \qquad \qquad \qquad \text{NORM_MIN_MAX}(\text{min}, \text{max}, (\text{Att2}, \text{Att3}), \text{Te})) \\
 & \qquad \qquad \qquad \equiv \\
 & \text{NORM_MIN_MAX}(\text{min}, \text{max}, (\text{Attr1}, \text{Att2}, \text{Att3}), \text{Te})
 \end{aligned}$$

Dimostrazione :

$$\begin{aligned}
 & \text{NORM_MIN_MAX}(\text{min}, \text{max}, (\text{Attr1}, \text{Att2}, \text{Att3}), \text{Te}) \\
 & \equiv \\
 & \text{Fnorm1_MM}(\text{TrovaMinAttuale}(\text{Te}, \text{Att1}), \\
 & \qquad \qquad \qquad \text{TrovaMaxAttuale}(\text{Te}, \text{Att1}), \\
 & \qquad \qquad \qquad \text{min}, \text{max}, \text{Att1}, \\
 & \qquad \qquad \qquad \text{Fnorm_MM}(\text{min}, \text{max}, (\text{Att2}, \text{Att3}), \text{Te}), \text{null}) \\
 & \equiv \{ \text{sia TE_23} = \text{NORM_MIN_MAX}(\text{min}, \text{max}, (\text{Att2}, \text{Att3}), \text{Te}) \\
 & \qquad \text{allora} \\
 & \qquad \qquad \text{Fnorm_MM}(\text{min}, \text{max}, (\text{Att2}, \text{Att3}), \text{Te}) == \text{TE_23} \\
 & \qquad \qquad \text{TrovaMinAttuale}(\text{Te}, \text{Att1}) == \text{TrovaMinAttuale}(\text{TE_23}, \text{Att1}) \\
 & \qquad \qquad \text{TrovaMaxAttuale}(\text{Te}, \text{Att1}) == \text{TrovaMaxAttuale}(\text{TE_23}, \text{Att1}) \\
 & \qquad \qquad \} \\
 & \text{NORM_MIN_MAX}(\text{min}, \text{max}, (\text{Att1}), \text{TE_23}) \\
 & \equiv
 \end{aligned}$$

NORM_MIN_MAX (min, max, (Att1),
NORM_MIN_MAX (min, max, (Att2,Att3), Te))

N.B.: In generale, per la normalizzazione non vale la proprietà di **distributività rispetto all'unione**, perché la trasformazione non è di tipo *locale*; a meno che non si conoscano in anticipo i parametri di normalizzazione, ossia nelle due varianti “locali” corrispondenti ai costrutti *NORM_MIN_MAX_L* e *NORM_Z_SCORE_L*.

Capitolo 5

Esempio: Preparazione dei log all'analisi di “clickstream”

Con l'ampliarsi della rete Internet e la crescente diffusione di siti che hanno finalità di natura economica, l'usabilità e l'accessibilità delle risorse in rete sono diventate più che mai dei fattori chiave di successo. Ciò ha portato allo sviluppo dei portali web adattivi, i quali riescono ad adattare dinamicamente la fornitura di servizi alle caratteristiche specifiche dei singoli utenti. Per raggiungere questo obiettivo è dunque indispensabile conoscere, e comprendere, l'interazione che questi hanno con il portale.

La sequenza delle risorse accedute da un utente nel corso della sua navigazione all'interno di un sito web è denominata *clickstream*, da cui il termine “Clickstream Analysis” per indicare l'attività di analisi di tali dati. A tale scopo, le sorgenti dei dati sono rappresentate da opportune strutture dati, ossia dai registri degli accessi (access log), in cui il web server archivia le informazioni relative alle richieste di accesso alle risorse che costituiscono un sito (pagine, contenuti multimediali, applicazioni). Generalmente tale struttura dati è mantenuta in uno o più files organizzati secondo delle specifiche standard: ad esempio l'NCSA Common Log Format (CLF), l'NCSA Extended Log Format (ECLF) e il W3C Extended Log File Format (ExLF). Nonostante le diverse configurazioni, alcune informazioni di accesso ad una risorsa sono sempre presenti:

- indirizzo IP o nome del client;
- data e orario dell'accesso;
- risorsa richiesta e protocollo utilizzato;
- esito dell'operazione;

- quantità di bytes trasmessi;
- identificativo della risorsa (URL) referente;
- user agent utilizzato (browser e piattaforma software).

Per tracciare in modo più accurato gli utenti si possono utilizzare degli opportuni moduli messi a disposizione dai web server, i quali contengono:

- l'identificativo che l'utente ha usato per accedere al client;
- l'identificativo del *cookie* usato per contrassegnare lo user agent dell'utente;
- l'identificativo che l'utente ha utilizzato per accedere al web server previa autenticazione.

Di seguito si riporta un estratto di log *ExLF*:

```
1 #Software: Microsoft Internet Information Services 5.0
2 #Version: 1.0
3 #Date: 2001-03-01 17:42:15
4 #Fields: c-ip date time cs-method cs-uri-stem sc-status
   sc-bytes cs-referer cs-user-agent
5 255.191.63.124 2001-03-01 08:30:20 GET /articles.html
   200 6169 http://www.alltheweb.com/cgi-bin/search
   "Mozilla/4.0 (compatible: MSIE 5.01; Windows NT)"
6 255.191.63.124 2001-03-01 08:30:22 GET /logo1.gif 200
   1900 http://www.alltheweb.com/ "Mozilla/4.0 (compatible:
   MSIE 5.01; Windows NT)"
7 ...
```

Tuttavia, i dati contenuti nei files di log sono di basso livello, e generalmente sono inadeguati a rappresentare direttamente l'interazione dell'utente con il portale. Vengono quindi applicate tecniche KDD alle informazioni di accesso, debitamente strutturate ed integrate in una fase preliminare di pre-processamento.

In questo capitolo proponiamo l'applicazione del formalismo DPL in riferimento al problema appena descritto. Si è presa in considerazione la parte iniziale di preparazione dei dati, che consiste nella pulizia ed organizzazione dei dati di log, al fine di predisporli per un'analisi *clickstream* successiva. In particolare, abbiamo preso spunto dal sistema

sviluppato nella tesi [Tesi5], in cui si è fornita una metodologia generale per trattare le diverse attività su cui si basa la costruzione ed il funzionamento di un portale web adattivo.

Il sistema preso come riferimento possiede una componente specifica per la preparazione dei log di accesso, per la successiva analisi di *clickstream*. L'approccio adottato è consistito nella stesura di un programma in Perl e nella specificazione di un insieme di files di configurazioni, attraverso i quali si possono personalizzare le operazioni di preparazione dei dati. Sono previsti tre tipi di files di configurazione, i quali contengono le specifiche di *data cleaning*, di *data aggregation* e di interfacciamento con la base di dati. Tali files sono organizzati in opportune sezioni e devono rispettare un formato prefissato.

Con il programma riportato di seguito si vuole proporre un approccio DPL al problema di preparazione dei dati di accesso (*log*) ad una successiva analisi. A tale scopo si sono presi in considerazione il programma Perl, assieme all'esempio di file di configurazione per il data cleaning *config.cfg*, al file di interfacciamento con la base di dati *db.cfg*, e agli esempi di specifiche di aggregazione fornite nel paragrafo 3.2.4 della tesi riferita. Tale script *Perl* prende in ingresso un file del registro degli accessi di un portale web, e produce un nuovo file di log ripulito in base alle specifiche contenute nel file di configurazione di data cleaning. Successivamente, esso effettua l'aggregazione dei dati, e solo alla fine, inserisce i dati di log nella base di dati, rispettando il formato ed i vincoli contenuti nel file di interfacciamento.

Noi proponiamo una soluzione, equivalente al sistema introdotto sopra, che consiste in un programma scritto nel formalismo sviluppato nella tesi presente. Si parte però dal presupposto che tutte le righe del file di log siano già inserite nella base di dati, e costituiscano l'unico attributo di una tabella *Accessi*. Tale tabella "grezza" subirà poi una serie di trasformazioni:

- si eliminano le tuple che non rappresentano un accesso, ovvero quelle che non soddisfano un formato prefissato;
- si effettuano trasformazioni di schema sulla tabella, strutturando le righe di log come tuple organizzate in una serie di attributi;
- si verificano i criteri di inclusione ed i criteri di esclusione attraverso tradizionali restrizioni in stile SQL, ma anche tramite controlli di pattern matching;

- si effettua la standardizzazione di alcuni attributi di accesso, attraverso l'applicazione delle regole di riscrittura, si identificano e si predispongono le sottocomponenti dei dati necessarie per le analisi di “clickstream”;
- si porta la base di dati nel formato specificato attraverso i files di configurazione.

L'ultimo passo di data cleaning nel sistema di riferimento riguarda la codifica hash delle urls in chiavi MD5. Tale codifica hash permette di risparmiare spazio nel nuovo file log che viene creato e di velocizzare la ricerca. Noi abbiamo trascurato questa elaborazione, immaginando che il sistema di gestione della base di dati offra la possibilità di creare in modo automatico una tabella hash su tutti gli attributi che contengono una *url*, potendo isolare, eventualmente, la sottocomponente che si intende codificare, attraverso l'utilizzo del pattern matching.

Una parte importante, e caratteristica, della preparazione all'analisi *clickstream* consiste nell'identificazione degli utenti e nella ricostruzione delle sessioni di navigazione. Tali questioni costituiscono un problema di difficile trattazione, perché la maggior parte degli utenti che accedono alla rete Internet sono anonimi (IP dinamico, postazioni multiple, accessi anonimi), e perché non è possibile stabilire una corrispondenza diretta tra client ed indirizzi IP. Queste operazioni si possono considerare di preprocessing, ma sono effettuate tramite algoritmi sviluppati appositamente, che dipendono dal tipo di informazione di cui si dispone (cookie, meccanismi di identificazione, ecc.), e non sono state implementate nel nostro programma.

Si è infine trascurato il completamento dei percorsi, problema dovuto alla presenza di *cache* e dei *proxies* nella rete. La soluzione proposta nel sistema di [Tesi5] è semplice, ma euristica. Nella figura seguente si mostra come vengono integrate le pagine mancanti.

page	referer	date
A	-	date1
B	A	date2
C	A	date3
D	A	date4



page	referer	date
A	-	date1
B	A	date2
<u>A</u>	<u>B</u>	<u>daten</u>
C	A	date3
<u>A</u>	<u>C</u>	<u>daten1</u>
D	A	date4

Figura 5.1: Integrazione di pagine mancanti

Nonostante l'operazione sia molto semplice, si è individuata una limitazione del linguaggio DPL per questo tipo di integrazione dei log. Non è infatti possibile inserire, in un punto interno alla tabella, una tupla, il cui contenuto dipende dall'elaborazione di una tupla adiacente. Se immaginiamo però di disporre di un linguaggio di script per MQL-DPL, in stile PL/SQL, che dispone di meccanismi di controllo del flusso e di un paradigma imperativo, allora si può facilmente implementare l'integrazione descritta sopra.

```
#####
#      Codice DPL per la preparazione dei dati di log      #
#####

#      Si suppone che tutte ( ! ) le righe del file di log siano inserite nella base di dati,
#      come unico attributo ( log ) di una tabella relazionale ( Accessi )

# Codice equivalente alla sezione Fields del file di configurazione config.cfg
# - - - marca le tuple che non rappresentano gli accessi del file di log, e le cancella
#      successivamente; il formato degli accessi è:
#
# ip=\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3} // indirizzo IP (4 numeri di al più 3 caratteri
#      -=\S+                               separati da puntino)
# user=\S+
# sysdate=\[.+\] // data e ora di sistema (racchiusa tra [ ])
# request="\.+\" // tipo di richiesta (GET, POST, etc)
# status=\S+ // stato (200, 100, 404...)
# contentlength=\S+ // dimensione in byte dei dati inviati
# host=\S+ // nome host
# referer="\.+\" // pagina referente (da cui si accede)
# cookie="\.+\" // cookie dell'utente
# useragent="\.+\" // tipo browser, vers S.O., etc

let logTable = Table2PPTable(Accessi)
in let logTable = APPLY RWRULES IN logTable TO log
  <<(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})\s(\S+)\s(\[.+\])\s(\".+\")\s(\S+)\s(\S+)\s(\S+)\s(\".+\")\s(\".+\")\s(\".+\")>>
  EXCEPTION codiceRimozione;
in let logTable = REMOVE ROW IN logTable
  WHERE EXCEPTION codiceRimozione

# - - - individuazione delle componenti di una riga di accesso, ed organizzazione
#      delle medesime in una tabella multi-attributo

in let logTable = SPLIT log IN logTable AT " " PUT INTO ip AND residuo
in let logTable = APPLY RWRULES IN logTable TO residuo
  <<\s(\S+)\s(.+)>> INTO $2;
in let logTable = SPLIT residuo IN logTable AT " " PUT INTO user AND residuo
in let logTable = APPLY RWRULES IN logTable TO residuo
  <<\s(.+)>> INTO $1;
in let logTable = SPLIT residuo IN logTable AT " "
  PUT INTO sysdate AND residuo
```

```

in let logTable =      APPLY RWRULES IN logTable TO residuo
                        <<\s(.+)>> INTO $1;
in let logTable =      SPLIT residuo IN logTable AT " "
                        PUT INTO request AND residuo
in let logTable =      APPLY RWRULES IN logTable TO residuo
                        <<\s(.+)>> INTO $1;
in let logTable =      SPLIT residuo IN logTable AT " " PUT INTO status AND residuo
in let logTable =      APPLY RWRULES IN logTable TO residuo
                        <<\s(.+)>> INTO $1;
in let logTable =      SPLIT residuo IN logTable AT " "
                        PUT INTO contentlength AND residuo
in let logTable =      APPLY RWRULES IN logTable TO residuo
                        <<\s(.+)>> INTO $1;
in let logTable =      SPLIT residuo IN logTable AT " " PUT INTO host AND residuo
in let logTable =      APPLY RWRULES IN logTable TO residuo
                        <<\s(.+)>> INTO $1;
in let logTable =      SPLIT residuo IN logTable AT " " PUT INTO referer AND residuo
in let logTable =      APPLY RWRULES IN logTable TO residuo
                        <<\s(.+)>> INTO $1;
in let logTable =      SPLIT residuo IN logTable AT " " PUT INTO cookie AND residuo
in let logTable =      APPLY RWRULES IN logTable TO residuo
                        <<\s(.+)>> INTO $1;
in let logTable =      SPLIT residuo IN logTable AT " "
                        PUT INTO useragent AND residuo
in let logTable =      REMOVE RESIDUO IN logTable

# Codice equivalente alla sezione Erase del file di configurazione config.cfg
# - - - si scartano i dati di log che non possiedono il cookie, quelli relativi a richieste di immagini
#       (*.img), ...

in let logTable =      APPLY RWRULES IN logTable TO request
                        BEGIN
                        <<.*rad>> INTO "criterio di esclusione verificato";
                        <<.*img>> INTO "criterio di esclusione verificato";
                        <<.*css>> INTO "criterio di esclusione verificato";
                        <<.*POST>> INTO "criterio di esclusione verificato";
                        <<.*js>> INTO "criterio di esclusione verificato";
                        END

in let logTable =      MARK log IN logTable
                        IF( ((ATTRIBUTE cookie) = ("-")) OR
                            ((ATTRIBUTE referer) = ("-")) ) WITH CODE codiceRimozione

in let logTable =      REMOVE ROW IN logTable
                        WHERE ((EXCEPTION codiceRimozione) OR
                               ( (ATTRIBUTE request) = ("criterio di esclusione verificato") ))

```

```

# Codice equivalente alla sezione IncludeOnly del file di configurazione config.cfg
# - - - sono mantenuti solo gli accessi relativi a richieste con successo e a pagine html

in let logTable =          REMOVE ROW WHERE ( (ATTRIBUTE status) != (200) )
in let logTable =          APPLY RWRULES IN logTable TO request
                           <<.*htm>> EXCEPTION codiceRimozione;
in let logTable =          REMOVE ROW IN logTable
                           WHERE EXCEPTION codiceRimozione

# Codice equivalente alla sezione Date del file di configurazione config.cfg
# - - - preparazione per una eventuale conversione in secondi della data

# formato dell'attributo data = (\d{2})\V(\w{3})\V(\d{4}):(\d{2}):(\d{2}):(\d{2}) .+
# dove le componenti della data sono:
#   day=1
#   month=2
#   year=3
#   hour=4
#   minute=5
#   second=6
#   wordmonth=Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec

# - - - marca i valori che non rispettano il formato richiesto e traduce in numeri la componente
#   mese della data, per un'eventuale conversione in secondi

in let logTable =          APPLY RWRULES IN logTable TO sysdate
                           BEGIN
                               <<(\d{2})\V(\w{3})\V(\d{4}):(\d{2}):(\d{2}):(\d{2}).+>>
                               EXCEPTION formatoDataNonConforme;
                               EVERY <<Jan>> INTO "01";
                               EVERY <<Feb>> INTO "02";
                               EVERY <<Mar>> INTO "03";
                               EVERY <<Apr>> INTO "04";
                               EVERY <<May>> INTO "05";
                               EVERY <<Jun>> INTO "06";
                               EVERY <<Jul>> INTO "07";
                               EVERY <<Aug>> INTO "08";
                               EVERY <<Sep>> INTO "09";
                               EVERY <<Oct>> INTO "10";
                               EVERY <<Nov>> INTO "11";
                               EVERY <<Dec>> INTO "12";
                           END

```



```

in let logTable =          APPLY RWRULES IN logTable TO referer
                           EVERY <<%( [A-Fa-f\d]{2} )>>
                           INTO DecoficaEsadecimale($1);

# Codice equivalente alle specifiche di aggregazione fornite nel paragrafo 3.2.4

# - - - scompone l'indirizzo url delle pagine referenti nelle sue componenti, e marca gli indirizzi
#       che non sono conformi al formato richiesto; serve nei siti a contenuto dinamico, nei quali
#       l'indirizzo della pagina referente determina il contenuto della pagina successiva

in let logTable =          APPLY RWRULES IN logTable TO referer
                           <<(\S+)/(\d),(\d+),(.*),00.html>>
                           EXCEPTION formatoURLNonConforme ;

in let logTable =          SPLIT referer IN logTable AT "/"
                           PUT INTO pathname AND residuo
in let logTable =          APPLY RWRULES IN logTable TO residuo
                           <<\/(.+)>> INTO $1;
in let logTable =          SPLIT residuo IN logTable AT ","
                           PUT INTO cacheable AND residuo
in let logTable =          APPLY RWRULES IN logTable TO residuo
                           <<,(.+)>> INTO $1;
in let logTable =          SPLIT residuo IN logTable AT "."
                           PUT INTO templateId AND residuo
in let logTable =          APPLY RWRULES IN logTable TO residuo
                           <<,(.+)>> INTO $1;
in let logTable =          SPLIT residuo IN logTable AT ":"
                           PUT INTO parameters AND residuo
in let logTable =          REMOVE residuo IN logTable

# - - - assegna ad ogni tupla di log la chiave che indica la "provenienza" della richiesta,
#       in base al valore dell'attributo pathname

in let logTable =          APPLY RWRULES IN logTable TO pathname
                           BEGIN
                           <<(VHome.*)>> INTO $1"#home";
                           <<(Vhomepage.* )>> INTO $1"#home";
                           <<(VnewsVNotizia.* )>> INTO $1"#notizie";
                           <<(VchefareVcalendario.* )>> INTO $1"#eventi";
                           <<(VchefareVEditEvent.* )>> INTO $1"#modificaevento";
                           <<(VchefareVDeleteEvent.* )>> INTO $1"#eliminaevento";
                           <<(VchefareVSearchEvent.* )>> INTO $1"#cercaeventi";
                           END

```


Capitolo 6

Conclusioni

Il lavoro di questa tesi è consistito nell'estensione del sistema KDDML-MQL in modo da supportare lo stadio della preparazione dei dati all'interno del processo KDD.

Preso visione delle componenti già integrate nel sistema, in particolare del linguaggio per interrogazioni di data mining MQL (Mining Query Language), si è deciso di progettare le nuove funzionalità richieste attraverso l'estensione di MQL. In questo modo, innanzitutto le diverse componenti sono integrate in modo uniforme, ma, cosa ancora più importante, nel loro utilizzo è facilitata l'interattività e l'iterazione tra lo stadio di data mining e quello del preprocessing.

Il prodotto del lavoro svolto è dunque un formalismo algebrico, denominato DPL (Data Preparation Language), il quale consente di formulare le trasformazioni di preprocessing come un processo di interrogazione dei dati.

Sono innanzitutto state fornite le primitive DPL, in seguito ad un'analisi delle problematiche che si pongono nei diversi momenti della preparazione dei dati durante un processo KDD. Si è inoltre riconosciuta la necessità di estendere il sistema con un meccanismo di marcatura, il quale si è rivelato utile per affrontare diverse questioni in fase di pulizia dei dati, così come può essere utilizzato come strumento di analisi. Si noti, che tramite la marcatura delle eccezioni che si verificano durante l'elaborazione di una trasformazione, viene realizzato anche un certo grado di interattività.

La definizione formale del linguaggio DPL è costituita da una semantica operativa, conforme a quella adottata nella specifica MQL. Il modello di astrazione è anche in questo caso la *sequenza*, e le primitive del linguaggio sono quindi viste come degli operatori derivati da quelli elementari su tali strutture di base.

Il processo KDD, ed in particolare la fase di preprocessing, non sono lineari. Anche se una buona progettazione delle primitive può ridurre il numero dei cicli, persiste

comunque la natura iterativa dell'intero processo di estrazione di conoscenza. Questa considerazione sottolinea l'importanza di integrare uno strumento di preparazione dei dati nel modo più uniforme possibile con gli strumenti di data mining. Nel nostro caso ha motivato la scelta di sviluppare il linguaggio DPL come un'estensione MQL. Questo approccio fornisce inoltre i presupposti migliori per adottare in futuro le strategie di *pulizia dei dati* basate su data mining.

L'innovazione del formalismo proposto in questa tesi sta, a nostro avviso, nell'aver riunito funzionalità molto diverse, offerte singolarmente da strumenti ed ambienti differenti, e di avere definito una semantica uniforme per tali funzionalità. In questo modo, adottando una rappresentazione univoca tra le tecniche di preparazione dei dati e quelle di data mining, si è raggiunto l'obiettivo di poter combinare tra loro, ed alternare, i diversi operatori, mantenendo così la proprietà di chiusura e la natura algebrica, che erano caratteristiche di MQL.

Vale inoltre la pena notare come, attraverso le verifiche di pattern matching previste nel linguaggio, non solo sia supportata la riscrittura di stringhe in fase di pulizia dei dati, ma, ovunque siano previste delle condizioni, è anche possibile effettuare un confronto di "similitudine".

Il lavoro futuro potrebbe consistere nell'estensione in varie direzioni del presente lavoro:

- implementare ed integrare DPL all'interno del sistema KDDML-MQL, e successivamente, effettuare la sperimentazione e l'ottimizzazione delle interrogazioni;
- stabilire una metafora grafica per il processo di estrazione di conoscenza, e fornire un "linguaggio visuale" di programmazione per applicazioni KDD, corrispondente a quello testuale MQL-DPL;
- estendere DPL con ulteriori primitive di preparazione dei dati;
- proporre un linguaggio integrato nello stile PL/SQL del sistema ORACLE, che estenda il linguaggio MQL-DPL con costrutti di controllo di tipo

generale, quali il *for*, l'*if*, il *while*, e consenta anche di memorizzare e denotare, attraverso l'astrazione funzionale, una sequenza di operazioni di preprocessing e di data mining;

- studiare delle strategie di *data cleaning* che coinvolgono algoritmi di data mining, e proporre una metodologia e degli esempi di applicazione nel sistema KDDML. Alcuni modelli di conoscenza potrebbero infatti fornire un supporto euristico per la rilevazione di dati “sporchi”, come ad esempio i clusters per gli *outliers* e le regole di associazione per i duplicati non evidenti, mentre gli alberi di predizione fornirebbero un valore stimato per i valori mancanti.

Ringraziamënc¹

Dan da düt urëssi ringrazié mi relatur, l'Prof. Franco Turini, pur súa costanta disponibilité, pur la crëta ch'al à albü te me, y pur ester stè bun da me menè cun savëi intratan chësc laûr.

Giulan incè al Dr. Salvatore Ruggieri pur l'material y pur i ütli sujerimënc che al m'à dè.

N giulan sinzâr va spo a mia familia, che m'á tres dè, de vigni vers, n sterch sostëgn; giulan tati y mama, che m'és prüma de düt dè la poscibilité de jì inant cun i stüdi, y Heino, sënza de tè n'i éssi bunamënter nia stlüc jö.

I urëss incè ringrazié mi amisc, dan da düt la Babi y la Stefi, da chëles ch'i à cíafè n apoj, intratan chisc agn, tles plö desvalies circostanzes.

Ala fin, n ringraziamënt particular va al Antonello, ch'm'à indô n iade daidé dër tröp: giulan pur tò bogn consëis, pur l'incorajamënt, mo incè pur m'avëi fat punsè a val' d'ater vigni tan.²

¹ Ho voluto scrivere i ringraziamenti nella mia madrelingua, il *ladino*.

² Innanzitutto vorrei ringraziare il mio relatore, il Prof. Franco Turini, per la sua costante disponibilità, per la fiducia dimostratami, e per la bravura con cui ha saputo guidarmi durante questo lavoro.

Ringrazio anche il Dott. Salvatore Ruggieri per i suggerimenti e per il materiale che mi ha fornito.

Un sincero ringraziamento va poi alla mia famiglia, da cui ho sempre avuto un forte sostegno: grazie papà e mamma, prima di tutto, di avermi dato la possibilità di proseguire gli studi all'università, e Heino, senza di te non li avrei probabilmente portati a termine.

Vorrei inoltre ringraziare i miei amici, soprattutto Barbara e Stefanie, da cui ho ricevuto appoggio nelle circostanze più diverse durante questi anni.

Infine, ad Antonello spetta un grazie particolare, per avermi saputo aiutare in diverse occasioni, con consigli utili, efficaci incoraggiamenti, e le indispensabili distrazioni.

Appendice A:

Le operazioni elementari sulle sequenze¹

1. Estrazione dal G_OBJ

Questa operazione effettua l'estrazione degli oggetti che compongono il G_OBJ o degli elementi associati, nel G_OBJ, ad un nome nella G_DECL per quel tipo. L'operazione prende in ingresso:

1. Il tipo di G_OBJ al quale deve essere applicata
2. Il nome dell'elemento o la specifica del tipo che deve essere estratto
3. Il G_OBJ al quale deve essere applicata

Restituisce la sequenza degli elementi estratti.

Segnatura:

$$\varepsilon : \text{Tipo}, \text{Nome}, \text{G_OBJ} \rightarrow \text{sequenza}$$

Esempio

Si consideri il G_OBJ Tabella: $\mathbf{T} \equiv \mathbf{Tabella\#(A_1, A_2), 2, ((V_1, V_2); (V_3, V_4))\#}$. Applichiamo "ε" con l'intento di estrarre, da tale G_OBJ, sia l'oggetto che lo compone che ogni nome nelle G_DECL per Tabella:

$$\varepsilon(\text{Tabella}, \text{Tabella}, T) \equiv (((A_1, A_2), 2, ((V_1, V_2), (V_3, V_4))))$$

$$\varepsilon(\text{Tabella}, \text{Attributi}, T) \equiv ((A_1, A_2))$$

$$\varepsilon(\text{Tabella}, \text{NumAttr}, T) \equiv (2)$$

$$\varepsilon(\text{Tabella}, \text{Valori}, T) \equiv (((V_1, V_2), (V_3, V_4)))$$

Se il G_OBJ Tabella considerato fosse stato vuoto: $\mathbf{T} \equiv \mathbf{Tabella\#(A_1, A_2), 2, null\#}$, l'applicazione di "ε" nei casi sopra riportati avrebbe restituito:

$$\varepsilon(\text{Tabella}, \text{Tabella}, T) \equiv (((A_1, A_2), 2, null))$$

¹ Tratto da [Tesi2]

$$\varepsilon(\text{Tabella}, \text{Attributi}, T) \equiv ((A_1, A_2))$$

$$\varepsilon(\text{Tabella}, \text{NumAttr}, T) \equiv (2)$$

$$\varepsilon(\text{Tabella}, \text{Valori}, T) \equiv \text{null}$$

Gli esempi riportati possono essere estesi per ogni altro tipo di G_OBJ: “ ε ” applicata al tipo, estrae la sequenza associata al tipo stesso e, applicata rispetto ad un nome nelle G_DECL per quel tipo, estrae la sequenza di elementi riferita da tale nome.

2. Costruzione di un G_OBJ

Questa operazione costruisce oggetti di un certo tipo seguendo la G_DECL associata al tipo stesso. L’operazione prende in ingresso:

1. Il tipo del G_OBJ da costruire
2. Una sequenza di elementi associata al tipo da costruire

Restituisce il G_OBJ costruito.

Segnatura

$$\gamma: \text{Tipo}, \text{sequenza} \rightarrow G_OBJ$$

Esempio

Supponiamo di avere la sequenza: $s \equiv ((A, 1, ((I_1, I_2), I_3, (I_4, I_5, I_6))))$ da cui vogliamo costruire un G_OBJ Tabella; per farlo si applica l’operazione “ γ ” come segue:

$$\gamma(\text{Tabella}, s)$$

Questa restituisce il G_OBJ **Tabella#A,1,((I₁,I₂);I₃;(I₄,I₅,I₆))#**; che rappresenta una Tabella con un solo attributo e composta da 3 tuple.

Così vale per ogni altro tipo di G_OBJ: “ γ ” costruisce il G_OBJ in accordo con la G_DECL per quel tipo, considerando come oggetti gli elementi identificati dalle sottosequenze più esterne.

3. Estrazione da una sequenza

Questa operazione effettua l'estrazione di un elemento che si trova in una determinata posizione all'interno di una sequenza. L'operazione prende in ingresso:

1. La sequenza di elementi
2. La posizione dell'elemento da estrarre

Restituisce l'elemento estratto.

Segnatura:

$\delta: \text{sequenza, posizione} \rightarrow \text{elemento}$

Esempio.

Supponiamo di avere la seguente sequenza di elementi:

$$s \equiv ((A_1, A_2), A_3, (A_4, A_5))$$

da cui vogliamo estrarre l'elemento in posizione due; per far questo si applica “ δ ” come segue:

$$\delta(s, 2)$$

Si ottiene l'elemento “ A_3 ”.

Se “ δ ” viene applicata rispetto ad una posizione che non esiste nella sequenza, restituisce l'elemento vuoto:

$$\delta(s, n) \equiv \text{null} \quad \text{per } n \geq 4$$

Se “ δ ” viene applicata rispetto ad una sequenza vuota, restituisce l'elemento vuoto:

$$\delta(\text{null}, n) \equiv \text{null} \quad \text{per } n = 1, 2, \dots$$

Se “ δ ” viene applicata rispetto ad un elemento che non è una sequenza, restituisce l'elemento vuoto:

$$\delta(A_3, n) \equiv \text{null} \quad \text{per } n = 1, 2, \dots$$

4. Cancellazione di un elemento da una sequenza

Questa operazione effettua la cancellazione di un elemento da una sequenza di elementi.

Prende in ingresso:

1. La sequenza di elementi
2. Un elemento

Restituisce una sequenza di elementi.

Segnatura:

$$\mu : \text{sequenza, elemento} \rightarrow \text{sequenza}$$

Esempio.

Consideriamo la sequenza:

$$s \equiv ((A_1, A_2), A_3, (A_4, A_5))$$

rispetto alla quale vogliamo cancellare l'elemento "A₃":

$$\mu \equiv (s, A_3) \equiv ((A_1, A_2), (A_4, A_5))$$

Se "μ" viene applicata rispetto ad un elemento che non è nella sequenza, restituisce la sequenza stessa:

$$\mu(s, (A_6, A_7)) \equiv s$$

Se "μ" viene applicata rispetto ad una sequenza vuota, restituisce la sequenza vuota:

$$\mu \equiv (\text{null}, e) \equiv \text{null} \quad \forall e$$

Se "μ" cancella l'unico elemento di una sequenza, restituisce la sequenza vuota:

$$\mu \equiv (((A_1, A_2)), (A_1, A_2)) \equiv \text{null}$$

5. Restituzione di una sequenza

Questa operazione effettua l'estrazione di una sequenza di elementi da una sequenza.

Prende in ingresso:

1. Una sequenza di elementi
2. Una sequenza di posizioni

Restituisce la sequenza di elementi che si trovano nelle posizioni prese come secondo parametro all'interno di ogni elemento della sequenza, presa come primo parametro.

Segnatura:

$$\rho : \text{sequenza}, \text{sequenza} \rightarrow \text{sequenza}$$

Esempio.

Supponiamo di avere la sequenza:

$$s \equiv ((A_1, A_2, A_3), (A_4, (A_5, A_6), A_7), (A_8, A_9, A_{10}))$$

da cui vogliamo estrarre la sequenza di elementi che si trovano in posizione "1" rispetto ad ogni elemento di "s"; per far questo si usa " ρ " come segue:

$$\rho(s, (1)) \equiv (A_1, A_4, A_8)$$

Se da "s" vogliamo estrarre la sequenza di elementi che si trovano in posizione "2" e "3" useremo:

$$\rho(s, (2, 3)) \equiv ((A_2, A_3), ((A_5, A_6), A_7), (A_9, A_{10}))$$

Se " ρ " viene applicato rispetto ad una posizione non presente in nessun elemento della sequenza che prende come primo parametro o alla posizione "0", restituisce la sequenza vuota:

$$\rho(s, (n)) \equiv \text{null per } n \geq 4$$

$$\rho(s, (i, n)) \equiv \text{null per } 1 \leq i \leq 3, n \geq 4$$

$$\rho(s, (0)) \equiv \text{null } \forall s$$

Se " ρ " viene applicata ad una sequenza vuota, restituisce una sequenza vuota:

$$\rho(\text{null}, (n)) \equiv \text{null per } n = 1, 2, \dots$$

$$\rho(s, \text{null}) \equiv \text{null } \forall s$$

6. Aggiunta di elementi ad una sequenza

Questa operazione esegue un “append” di elementi rispetto ad una sequenza. Prende in ingresso:

1. Una sequenza di elementi
2. Una sequenza di elementi

Restituisce una sequenza composta dagli elementi di entrambe le sequenze che prende come input.

Segnatura

$$\alpha: \text{sequenza}, \text{sequenza} \rightarrow \text{sequenza}$$

Esempio.

Consideriamo le due sequenze:

$$s \equiv (A_1, A_2, (A_3, A_4)), s_1 \equiv (A_6, (A_5, (A_8, A_9)))$$

a cui applichiamo l’operazione “ α ”:

$$\alpha(s, s_1) \equiv (A_1, A_2, (A_3, A_4), A_6, (A_5, (A_8, A_9)))$$

Se “ α ” viene applicata a due sequenze vuote, restituisce una sequenza vuota:

$$\alpha(\text{null}, \text{null}) \equiv \text{null}$$

Se una delle due sequenze prese da “ α ” come input è vuota, l’operazione restituisce la sequenza non vuota:

$$\alpha(\text{null}, s) \equiv \alpha(s, \text{null}) \equiv s \quad \forall s$$

7. Ricerca di un elemento all’interno di una sequenza

Trova la posizione di un elemento all’interno di una sequenza priva di sottosequenze.

Prende in ingresso:

1. Una sequenza di elementi
2. Un elemento

Restituisce la posizione in cui si trova l'elemento, preso come secondo parametro, all'interno della sequenza, presa come primo parametro.

Segnatura:

$$\tau: \text{sequenza, elemento} \rightarrow \text{posizione}$$

Esempio.

Supponiamo di avere la sequenza di elementi:

$$s \equiv (A_1, A_2, A_3, A_6, A_9)$$

e di essere interessati a conoscere la posizione dell'elemento "A₆"; per far questo applichiamo "τ" nel seguente modo:

$$\tau(s, A_6) \equiv 4$$

L'applicazione di "τ" ad una sequenza vuota restituisce la posizione "0":

$$\tau(\text{null}, e) \equiv 0 \quad \forall e$$

Se l'elemento preso come secondo parametro non è presente nella sequenza "τ" restituisce la posizione "0":

$$\tau(s, A_7) \equiv 0$$

8. Restrizione degli elementi di una sequenza

Questa operazione effettua la restrizione degli elementi appartenenti ad una sequenza.

Prende come ingresso:

1. Una condizione
2. Una posizione
3. Una sequenza di elementi

Restituisce la sequenza composta dagli elementi che verificano la condizione, presa come primo parametro, nella posizione presa come secondo parametro.

La condizione può riguardare:

- Una limitazione del valore che può essere assunto da un elemento. La condizione è espressa come un operatore di confronto seguito da una costante, appartenente ad dominio dei vaolori dell'elemento a cui la condizione viene applicata, che indica la limitazione: $\leq c, \geq c, = c, \dots$
- La presenza o meno di un valore in riferimento ad un elemento. La condizione è espressa esplicitando il valore dell'elemento o esplicitando tale valore ma preceduto dal simbolo “ \neg ”:
 I oppure $\neg I$. Questo tipo di condizione viene usato quando l'elemento può assumere più valori in riferimento allo stesso oggetto.
- Il numero di valori che possono essere assunti da un elemento. La condizione è espressa come un operatore di confronto seguito dal simbolo “ e ” e da una costante che indica il numero di valori: $\leq_e c, \geq_e c, =_e c, \dots$

Segnatura:

$$\sigma: \text{condizione, posizione, sequenza} \rightarrow \text{sequenza}$$

Esempio.

Supponiamo di avere la sequenza:

$$s \equiv ((A_1, A_2, A_3), (A_4, A_6, A_3), (A_5, A_4, (A_8, A_9)))$$

di cui siamo interessati agli elementi che in posizione “3” non hanno il valore “ A_3 ”; applicheremo “ σ ” come segue:

$$\sigma(\neg A_3, 3, s) \equiv ((A_5, A_4, (A_8, A_9)))$$

Se, invece, siamo interessati agli elementi che in posizione “3” hanno il valore “ A_3 ” applichiamo “ σ ” come segue:

$$\sigma(A_3, 3, s) \equiv ((A_1, A_2, A_3), (A_4, A_6, A_3))$$

Se vogliamo gli elementi di “s” che in posizione “3” hanno più di un elemento usiamo:

$$\sigma(>_e 1, 3, s) \equiv ((A_5, A_4, (A_8, A_9)))$$

Se vogliamo gli elementi che in posizione “1” sono “uguali” ad “A₁” usiamo:

$$\sigma(=A_1, 1, s) \equiv ((A_1, A_2, A_3))$$

Se “ σ ” viene applicata rispetto ad una posizione non presente in nessun elemento, restituisce la sequenza vuota:

$$\sigma(cond, 4, s) \equiv null \quad \forall cond$$

Se “ σ ” viene applicata ad una sequenza vuota, restituisce una sequenza vuota:

$$\sigma(cond, n, null) \equiv null \quad \forall cond, n$$

Le condizioni espresse al livello di astrazione dell’utente saranno, in generale, più complesse di quelle che possono essere espresse a questo livello. E’ comunque sempre possibile ottenere un’equivalenza fra le condizioni espresse rispetto ai due livelli di astrazione:

1. **A θ c, c θ A** con θ operatore di confronto, **c** costante in dom(A) e **A** attributo in T (Tabella). Può essere espressa come:

$$\sigma(\theta c, \mathcal{A}(sequenza_attributi, A), sequenza_valori)$$

2. **A in nome** (**\neg A in nome**) con **A** possibile valore per **nome** e **nome** uno degli elementi che compongono l’oggetto a cui si applica la condizione. Può essere espressa come:

$$\sigma(A, pos_nome, sequenza_valori)$$

$$\sigma(\neg A, pos_nome, sequenza_valori)$$

3. **<elem in> nome θ c** con **nome** uno degli oggetti a cui si applica la condizione o un attributo in T (Tabella), θ operatore di confronto, **c** costante. Può essere espressa come:

$$\sigma(\theta_e c, pos_nome, sequenza_valori)$$

4. Se ϕ , ψ sono condizioni allora lo sono anche $\phi \vee \psi$, $\phi \wedge \psi$, $\neg \phi$. La condizione $\phi \vee \psi$ può essere espressa come:

$$\alpha(\sigma(\text{cond}_{\phi} \text{pos}_{\phi} \text{sequenza_valori}), \sigma(\text{cond}_{\psi} \text{pos}_{\psi} \text{sequenza_valori}), \sigma(\neg \text{cond}_{\phi} \text{pos}_{\phi} \text{sequenza_valori}))$$

La condizione $\phi \wedge \psi$ può essere espressa come:

$$\sigma(\text{cond}_{\phi} \text{pos}_{\phi} \sigma(\text{cond}_{\psi} \text{pos}_{\psi} \text{sequenza_valori}))$$

La condizione $\neg \phi$ può essere espressa come:

$$\sigma(\neg \text{cond}_{\phi} \text{pos}_{\phi} \text{sequenza_valori})$$

Bibliografia

[ACM99] S. Abiteboul, S. Clue, T. Milo, P. Mogilevsky, J. Simeon: “*Tools for Data Translation and Integration*”. In Special Issue on Data Transformation, IEEE Tech. Bull. Data Engineering 22(1):3-8,1999.

[ADT00] P. Alcamo, F. Domenichini and F. Turini “*An XML Based Environment in Support of the Overall KDD Process*”. Dipartimento di Informatica, Università di Pisa, 2000.

[AGO97] A. Albano, G. Ghelli, R. Orsini: “*Basi di dati relazionali e a oggetti*”. Zanichelli, 1997.

[BDM] F. Giannotti: Lucidi del corso di Data Mining (2002/2003):
<http://www-kdd.cnuce.cnr.it/bdm>.

[BDMP] Giuseppe Manco: “*Preparazione di Dati per Data Mining*” (Lucidi seminario): <http://www.isi.cs.cnr.it/isi/manco/dm>.

[CDH99] J. Clear, D. Dunn, B. Harvey, M. Heytens, P. Lohmanm, A. Mehta, M. Melton, L. Rohrberg, A. Savasere, R. Wehrmeister, M. Xu: “*NonStop SQL/MX Primitives for Knowledge Discovery*” . Compaq Computer Corporation, 1999.

[Clem] Clementine - Data Mining, Predictive models:
<http://www.spss.com/SPSSBI/Clementine>.

[ETL] Data extraction, transformation, and loading tools (ETL):
<http://www.dwinfocenter.org/clean.html>.

[FPS96] U. Fayyad, G. Piatetsky-Shapiro and P. Smyth, “*From Data Mining to Knowledge Discovery: An Overview*” in *Advances in Knowledge Discovery in Data Mining*, U. Fayyad, G. Piatetsky-Shapiro and P. Smyth, R. Uthurusamy, 1996, pp 1-36.

[GFS00] H. Galhardas, D. Florescu, D. Shasha, E. Simon, J.P. Matsumoto, C.A. Saita: “*AJAX: An Extensible Data Cleaning Tool*” INRIA, France. In SIGMOD, 2000: <http://cosmos.inesc.pt/~hig/ajax.html>.

[GFSS00] H. Galhardas, D. Florescu, D. Shasha, E. Simon: “*Declaratively cleaning your data using AJAX*”. In Journées Bases de Données, Oct 2000: <http://citeseer.nj.nec.com/309494.html>.

[HFKWZ] J. Han, Y. Fu, K. Koperski, W. Wang, O. Zaiane: “*DMQL: A Data Mining Query Language for Relational Databases*”. DMKD-96 (SIGMOD-96 Workshop on KDD), Montreal, Canada.

[HK00] J. Han, M. Kamber “*Data Mining: Concepts and Techniques*”. Morgan Kaufman, 2000.

[HS98] M. A. Hernandez, S. J. Stolfo: “*Real-World Data is Dirty: Data Cleansing and the Merge/Purge Problem*”. *Data Mining and Knowledge Discovery* 2(1) :9-37,1998.

[IV99] T. Imielinsky and A. Virmani “*MSQL: A Query Language for Database Mining*”, Rutgers University, Department of Computer Scienze, New Brunswick, USA, 1999.

[JDR] Paul Jermyn, Maurice Dixon and Brian J. Read: “*Preparing Clean Views of Data for Data Mining*”: http://www.ercim.org/publication/ws-proceedings/12th-EDRG/EDRG12_JeDiRe.pdf.

[KDD] Data Mining & KDD: <http://www.lido.dist.unige.it/LavoriStudenti/DM/start-ita.htm>.

- [**KDDlab**] The Pisa KDD Laboratory: <http://www-kdd.cnuce.cnr.it/default.htm>.
- [**KDDML**] KDDML-MQL Archive: <http://kdd.di.unipi.it/kddml/>.
- [**M00**] A. E. Monge : “*Matching Algorithm within a Duplicate Detection System*”.
IEEE Techn. Bulletin Data Engineering 23(4),2000.
- [**MMart**] MiningMart Research Project – Datawarehouse Mining: <http://mmart.cs.uni-dortmund.de/research/index.html>.
- [**MP98**] R. Meo, G. Psaila “*An Extension to SQL for Mining Association Rules*”, Data Mining and Knowledge Discovery, 2(2):195-224 (1998).
- [**MZ97**] T. Morzy, M. Zakrewicz: “*SQL-like language for database mining*”.
Proceedings of the first East-European Symposium of Advances in Databases and Information System (AD-BIS’97), St. Petersburg, September 2-5, 1997.
- [**Pyle99**] Dorian Pyle: “*Data Preparation for Data Mining*”.Morgan Kaufman, 1999.
- [**Quass99**] Dallon Quass: “*A Framework for Research in Data Cleaning*”. DRAFT-1999, Brigham Young University: <http://cosmos.inesc.pt/~hig/quass.cleaningII.pdf>.
- [**RD**] Erhard Rahm, Hong Hai Do: “*Data Cleaning: Problems and Current Approaches*”. University of Leipzig, Germany:
http://www.witi.cs.uni-magdeburg.de/iti_db/lehre/dw/paper/data_cleaning.pdf.
- [**RE1**] Robert D. Cameron: “*Perl Style Regular Expressions in Prolog*”. CMPT 384 Lecture Notes, November 29 - December 1, 1999:
<http://www.cs.sfu.ca/people/Faculty/cameron/Teaching/384/99-3/regexp-plg.html>.
- [**RE2**] Documentation on Package java.util.regex (Java 2 Platform SE v1.4.1) :
<http://java.sun.com/j2se/1.4.1/docs/api/java/util/regex/package-summary.html>.

[RH01] Vijayshankar Raman, Joseph M. Hellerstein: “*Potter’s Wheel: An Interactive Data Cleaning System*”. University of California at Berkeley: <http://www.vldb.org/conf/2001/P381.pdf>.

[SA95] R. Srikant, R. Agrawal: “*Mining Generalized Association Rules*”. Proc. 21st VLDB conf., 1995.

[SKB00] Lorenza Saitta, Jörg-Uwe Kietz, Giuseppe Beccari: “*Specification of Pre-Processing Operators Requirements*” (2000): <http://citeseer.nj.nec.com/575095.html>.

[SQLServer] Microsoft SQL Server 2000, On-Line Documentation.

[Tesi1] P. Alcamo, F. Domenichini “*Un ambiente basato su XML per l’estrazione di conoscenza*” (tesi) relatore Prof Franco Turini, 2000.

[Tesi2] M. Baglioni: “*MQL: Una proposta di linguaggio per Data Mining*” (tesi) relatore Prof. Franco Turini, 2001.

[Tesi3] D. Bruno: “*Estensione e sperimentazione di un sistema di knowledge discovery basato su XML*” (tesi) relatori Prof. Franco Turini e Prof Salvatore Ruggieri, 2001.

[Tesi4] A. Romei: “*Implementazione di un Query Language per Knowledge Discovery*” (tesi) relatori Prof. Franco Turini e Prof Salvatore Ruggieri, 2002.

[Tesi5] Luca Manganelli: “*Un sistema per il preprocessing di log di Web Server*” (tesi) relatori Prof. Paolo Bancarella, Prof. Salvatore Ruggieri e Prof. Umberto Ferrara, 2002.

[WEKA] Waikato Environment for Knowledge Analysis:
<http://www.cs.waikato.ac.nz/ml/weka>.

[WS92] Larry Wall, Randal L. Schwartz: “*Programming Perl*”. O’Reilly & Associates, Inc., 1992.