ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
SEDE DI CESENA
FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
CORSO DI LAUREA IN SCIENZE DELL'INFORMAZIONE

Gestione del versioning in sistemi di data warehouse

Tesi di Laurea in

Sistemi Informativi

Relatore

Chiar.mo Prof. Matteo Golfarelli

Presentata da

Alessandro Ronchi

Sessione I

Anno Accademico 2004/05

Alla mia famiglia
ed a tutte le persone
capaci di voler bene

Chi riceve un'idea da me,
ricava conoscenza senza diminuire la mia;
come chi accende la sua candela con la mia,
riceve luce senza lasciarmi al buio.
(Thomas Jefferson)

Indice

Elenco delle figure

Capitolo 1

Introduzione

L'inizio è la parte più importante di un lavoro.
(Platone)

I data warehouse sono database specializzati per l'uso nel campo della business intelligence, finalizzati al supporto delle decisioni in ambito aziendale.

L'uso di sistemi di data warehouse è aumentato rapidamente all'interno del mondo industriale nell'ultimo decennio, e questo successo è dovuto essenzialmente al contributo che hanno saputo apportare all'efficienza ed all'efficacia dei processi decisionali all'interno del dominio aziendale e scientifico. Questa ampia diffusione è stata sostenuta dagli importanti risultati della ricerca in questo campo, volti ad incrementare le performance delle interrogazioni e la qualità dei dati, così come dal rapido sviluppo dei tool commerciali.

A differenza di quanto accade nei database operazionali, nei data warehouse i problemi che riguardano la storicizzazione delle informazioni hanno un impatto maggiore, dato che le interrogazioni coinvolgono periodi di tempo solitamente più lunghi. Diventa comune, infatti, il caso in cui venga richiesto di lavorare su insiemi di dati che corrispondono a lunghi intervalli di tempo, corrispondenti a definizioni intensionali potenzialmente disomogenee, progettate per soddisfare esigenze diverse. Il problema emerge ovviamente all'aumentare del tempo di vita del sistema, dato che le evoluzioni necessarie che non sono state gestite determineranno inevitabilmente un distacco maggiore tra la realtà e la sua rappresentazione all'interno del database,

con il rischio che questo diventi presto obsoleto ed inutilizzabile.

Fino ad ora la ricerca si è interessata principalmente ai cambiamenti del livello dei dati, prendendo in considerazione quindi le modifiche alle istanze delle gerarchie di aggregazione. Alcuni prodotti commerciali permettono effettivamente di gestire e tenere traccia dei cambiamenti dei dati e di supportare interrogazioni dei cubi basati su scenari temporali differenti.

A livello intensionale il problema è più complesso, poiché lo schema del data warehouse può cambiare per gestire l'evoluzione dei requisiti aziendali. Possono quindi rendersi necessarie nuove misure e proprietà, mentre altre possono con il tempo risultare obsolete ed inutili. Si può considerare come esempio il caso in cui si aggiunga un livello di dettaglio nella descrizione delle categorie dei prodotti, aggiungendo una sotto-categoria: lo schema di riferimento dovrà mutare per tenere in considerazione questa nuova necessità.

Le soluzioni possibili per questo problema possono essere suddivise in due categorie distinte: l'evoluzione ed il versioning degli schemi. La prima permette di modificare lo schema in base ai cambiamenti necessari, perdendo le definizioni precedenti, mentre la seconda permette di tenere traccia delle modifiche ed utilizzare le diverse definizioni in base al contesto della formulazione delle interrogazioni. Il versioning a livello degli schemi nei data warehouse fino a questo momento è stato solo parzialmente esplorato e non esistono sistemi commerciali di supporto al progettista che permettano di gestire i cambiamenti dei modelli.

Lo scopo di questa tesi è quello di analizzare il problema temporale nei data warehouse, di presentare un approccio per la gestione del versioning e di realizzare un prototipo di sistema per la modellazione degli schemi e la gestione delle versioni. Questo prototipo dovrà supportare la gestione delle history degli schemi concettuali ed il backporting assistito delle nuove modifiche alla versione corrente negli schemi aumentati delle versioni precedenti. In questo modo sarà quindi possibile evitare l'obsolescenza degli schemi concettuali, garantire una minore distanza tra i modelli e la realtà modellata, ed allo stesso tempo rendere disponibili i vecchi dati per le interrogazioni inter-schema che coinvolgono versioni temporali diverse della history degli schemi.

L'organizzazione di questo lavoro di tesi prevede una introduzione dello stato dell'arte delle basi di dati temporali, analizzato all'interno del secondo capitolo, ed un successivo approfondimento delle peculiarità del problema del tempo all'interno dei data warehouse, contenuto nel terzo capitolo. Nel quarto capitolo viene introdotto un approccio per la risoluzione del problema, mentre nel quinto viene mostrato il lavoro di progettazione del prototipo di gestione del versioning.

Capitolo 2

Basi di dati temporali

2.1 Problema dell'evoluzione degli schemi dei dati

Con il procedere del tempo, le necessità di gestione delle informazioni all'interno di un dominio applicativo possono cambiare anche considerevolmente. Basta pensare alle continue modifiche che sono necessarie a causa degli interventi di riorganizzazione aziendale e di razionalizzazione delle risorse, esempi tipici di una realtà che cambia e si adatta alle esigenze espresse per adattarsi ad un mercato sempre più competitivo e dai confini sempre meno definiti.

I processi aziendali si evolvono e la necessità di mantenere il minimo distacco possibile tra la realtà e la sua astrazione all'interno dei sistemi informativi è sempre maggiore, con le diverse problematiche connesse.

Sebbene le stime in questo campo differiscano, gli esperti sono concordi nell'affermare che il 50% o più degli sforzi dei progettisti e dei programmatori sono causati da modifiche del sistema dopo l'implementazione [Lie83], ed il compito di facilitare questi cambiamenti è reso più difficile se sono coinvolte grandi quantità di dati o di applicazioni. Secondo questa analisi, le modifiche ad un sistema informativo che coinvolgono cambiamenti nella struttura del database sono abbastanza frequenti. Come risultato, le modifiche agli schemi dei database sono un compito comune ed abbastanza problematico degli amministratori di database. Queste considerazioni sono molto importanti, sia dal punto di vista dei produttori di sistemi informativi, sia da quello degli utenti aziendali di questi sistemi.

I *database temporali* sono stati introdotti per gestire l'evoluzione temporale degli

oggetti modellati e per venire incontro a queste esigenze. Solitamente gestiscono due diverse dimensioni temporali: *transaction time*, che esprime il momento nel quale l'evento è registrato nel database, e *valid time*, che indica quando l'evento accade, è accaduto o è stimato che accada nel mondo reale. A seconda delle dimensioni temporali che essi supportano, i database temporali sono classificati in *monotemporali* (sia *transaction time* che *valid time*), *bi-temporali* o *Snapshot*. I database *transaction time* registrano tutte le versioni dei dati inseriti, cancellati o modificati in transazioni successive. I database *valid time* mantengono le versioni dei dati più recenti, ognuna relativa ad un distinto intervallo *valid time*. I database *bi-temporali* supportano sia il tempo delle transazioni che il valid time e quindi mantengono tutte le versioni valid time registrate in transazioni successive. I database *snapshot* non supportano direttamente la gestione del tempo: mantengono solamente l'ultima versione. I database nei quali coesistono relazioni di diversi formati temporali (*snapshot*, *valid time*, *bi-temporali*) sono chiamati *multi-temporali*.

L'applicazione di queste estensioni temporali sono state studiate per un buon numero di paradigmi di database, incluso quelli relazionali, quelli orientati agli oggetti e quelli deduttivi.

Nel campo dei database tradizionali, una modifica allo schema dei dati comporta immediatamente la sostituzione di quello precedente. I dati corrispondenti allo schema precedente sono persi, oppure mantenuti ed adattati al nuovo schema nel caso il database supporti lo *Schema Evolution*. In entrambi i casi, una porzione delle informazioni intensionali possono non essere più disponibili, insieme con i corrispondenti insiemi di informazioni estensionali.

Definizione 1 (Schema Modification) *Viene eseguita una modifica allo schema (*Schema Modification*) quando un DBMS permette di cambiare la definizione dello schema in un database popolato di dati*

Le modifiche agli schemi dei database hanno comunque bisogno dell'intervento del database administrator, in maniera più o meno pesante. In ogni caso è auspicabile che queste operazioni richiedano il minimo numero di interventi possibili affinché tutti i cambiamenti necessari vengano effettuati. I dettagli dell'implementazione dei cambiamenti dipendono in misura maggiore dal sistema sul quale si opera, piuttosto che sulle necessità dell'utente. E' chiaro, quindi, che l'attenzione dell'amministratore

Tabella 2.1: Un esempio di modifica del dominio

Staff id	**Posizione**	**Salario EU**
1	G55	33000
2	G56	37000
3	A05	45000
4	A09	65500
5	G51	32000

vada focalizzata sulle necessità del dominio sul quale vengono applicate le modifiche, piuttosto che sui dettagli dell'implementazione.

Queste considerazioni portano alla necessità di astrarre le singole operazioni necessarie alla gestione delle modifiche, e di gestirle in maniera semiautomatica attraverso l'uso di interfacce che semplifichino la modellazione concettuale e la propagazione dei cambiamenti.

2.1.1 Evoluzione dei domini

L'evoluzione di un dominio rappresenta una delle modifiche più semplici che possono essere apportate ad un modello di dati, ma rappresenta tuttora un problema di difficile risoluzione nella gestione degli schemi del database.

Prendiamo l'esempio mostrato nella tabella 2.1, che mostra un elenco di dipendenti, le relative posizioni ed i corrispondenti stipendi. Supponiamo che i codici che individuano la posizione vengano cambiati per necessità in nuovi codici basati su un nuovo dominio incompatibile con il precedente, ad esempio diventando codici basati su interi di quattro cifre. L'amministratore di database si trova di fronte a diversi problemi legati alla gestione dei dati esistenti:

- L'attributo del codice deve essere definito come alfanumerico, nonostante la nuova posizione sia definita interamente numerica?

- E' necessario inserire un nuovo attributo per mantenere i vecchi codici e, nel caso, per quanto tempo dovrà essere mantenuto? Come si rapporta questo vecchio attributo con il nuovo all'interno delle applicazioni?

- Come ci si comporta di fronte ai cambiamenti di posizione o gli impiegati in pensione per i quali non esiste nessun nuovo codice?

Il problema dell'evoluzione dei domini è fortemente associato all'espressività ed alla struttura dell'insieme dei tipi di dato utilizzati da un DBMS. L'approccio dell'SQL, basato sull'uso delle stringhe di caratteri, tipi numerici esatti e tipi numerici approssimati, richiede molti più interventi dell'amministratore di database rispetto ad un approccio nel quale sia disponibile un sistema di casting nello stile della programmazione in C, come ad esempio nell'SQL-92.

2.1.2 Evoluzione delle relazioni o delle classi

L'evoluzione delle relazioni e delle classi comprende la *definizione*, la *ridefinizione* e la *cancellazione* degli attributi, delle relazioni e delle classi, nel caso del paradigma dei database ad oggetti. Nel caso dei database ad oggetti in questo insieme di operazioni deve essere inclusa anche la modifica della *gerarchia delle classi*, nel caso si utilizzi una ereditarietà semplice, oppure del *grafo aciclico* (*lattice*) che rappresenta l'organizzazione delle classi, nel caso dell'ereditarietà multipla. All'interno del paradigma dei database temporali, invece, bisogna includere la *disattivazione* e la *riattivazione* degli attributi e delle relazioni/classi.

In letteratura esistono diverse proposte sulle modalità di gestione dei cambiamenti dello schema del database a livello delle relazioni e delle classi. Nell'ambito del paradigma ad oggetti un metodo consueto è quello di stabilire un insieme di invarianti per assicurare l'integrità semantica dello schema, ed un insieme di regole o di primitive per effettuare i cambiamenti dello schema ([LH90]). Nei database relazionali viene proposto un insieme di operazioni atomiche la cui composizione risulti consistente e porti a modifiche reversibili nella struttura del database ([ST82]).

In [RCR93] viene proposta una tassonomia per lo schema Versioning basata sul modello relazionale ed Entità Relazionale (ER). L'insieme di queste operazioni sono elencate nella tabella 2.2, suddivise per ogni categoria di evoluzione possibile.

Tabella 2.2: Una tassonomia per lo Schema Versioning basata sul modello relazionale

Tipo di evoluzione	**Operazioni**
Evoluzione del dominio e degli attributi	Espandere il dominio di un attributo
	Restringere il dominio di un attributo
	Cambiare il dominio di un attributo
	Aggiungere un attributo al database
	Rinominare un attributo
Evoluzione delle relazioni	Aggiungere una relazione
	Attivare una relazione
	Disattivare una Relazione
Assegnamenti Attributi-Relazioni	Aggiungere un attributo ad una relazione
	Disattivare un attributo
	Promuovere un attributo all'interno della chiave
	Rimuovere un attributo dalla chiave
	Dividere una relazione
	Partizionare una relazione
	Fare il join di due relazioni
	Fondere due relazioni
Supporto alle transazioni dello schema	Commit dello schema
	Rollback dello schema

Molte operazioni di modifica sugli schemi coinvolgono operazioni composte, quindi si suggerisce l'utilizzo di un meccanismo per il commit ed il rollback a livello dello schema (DDL), che dovrebbe essere separato dalle operazioni di commit e rollback a livello dei dati (DML). In aggiunta, le operazioni di commit a livello dei dati potrebbero non funzionare correttamente quando sono attive transazioni a livello dello schema. Se i dati da aggiornare possono non essere applicabili ad uno schema le cui modifiche non siano state ancora rese attive con il commit, non è corretto permettere gli inserimenti al di fuori dell'ambito della sessione corrente fino a quando i cambiamenti a livello di schema non sono stati eseguiti ed applicati. Questo porta alla necessità di completare la definizione e la popolazione degli attributi in una unica operazione atomica. L'algoritmo che segue, scritto in pseudo-codice, mostra un esempio della sequenza delle operazioni che possono essere eseguite come test per un programma che modifica e popola un attributo nella relazione di un database.

```
Aggiungi un attributo alla relazione;
Popola l'attributo con i dati;
Esegui il commit a livello dei dati;
Esegui i programmi di test;
if (i test sono corretti)
          { Esegui il commit sul livello dello schema; }
else
          { Esegui il rollback sul livello dello schema; }
endif
```

Il controllo attraverso i programmi di test e la disponibilità delle operazioni di commit e rollback permettono di arrivare ad uno stato consistente sia in presenza di una esecuzione corretta, sia nel caso non vada a buon fine, tornando allo stato precedente della relazione.

2.2 Soluzioni proposte

2.2.1 Evoluzione

In accordo con le definizioni fornite da [Rod95]:

Definizione 2 (Schema Evolution) *Viene applicato un sistema di* Schema Evolution *quando un DBMS facilita le modifiche dello schema del Database senza perdite dei dati esistenti.*

Questo è il caso più diffuso nei database tradizionali, quando non è necessario tenere una traccia delle modifiche effettuate, ed i dati esistenti possono essere tranquillamente adattati al nuovo schema del database, modificato secondo le esigenze attuali. Le informazioni intensionali ed estensionali che vengono perse sono considerate in questo caso poco rilevanti, e le interrogazioni vengono applicate tutte sull'ultimo schema. Nel suo caso più semplice, lo *Schema Evolution* non implica nessun supporto storico per lo schema, ma solo la possibilità di cambiare la definizione senza perdite di dati. Nella pratica, quindi, è sempre utile ed appropriato eseguire un salvataggio delle definizioni passate prima di effettuare modifiche allo schema. Questo approccio, che conta un indiscutibile vantaggio nella semplicità di gestione delle modifiche allo schema, può introdurre problemi nella compatibilità. Le applicazioni che si basano sugli schemi che sono stati modificati e non sono stati mantenuti, dovranno essere controllate e modificate per adattarsi alla nuova struttura, altrimenti possono cessare il loro funzionamento.

2.2.2 Versioning

Per supplire alla necessità di evitare perdita di informazioni, ed allo stesso tempo garantire il corretto funzionamento delle applicazioni che necessitano di lavorare su schemi multipli, lo *Schema Evolution* non è più sufficiente e diventa necessario il mantenimento di più di uno schema. Per questo motivo sono stati introdotti i concetti di *Schema Versioning* e *Schema Version*, a partire da [Rod95, JCE+94].

Definizione 3 (Schema Versioning) *Un DBMS supporta lo* Schema Versioning *quando permette l'accesso a tutti i dati, sia in maniera retrospettiva che prospettica, attraverso versioni che fungono da interfacce e che sono definite dall'utente.*

Anche nella sua forma più semplice, lo *Schema Versioning* richiede che sia mantenuta una storia dei cambiamenti per la gestione delle definizioni precedenti dello schema.

Una differenza significativa tra l'evoluzione ed il versioning è l'abilità per gli utenti di identificare dei punti e momenti di stabilità delle definizioni degli schemi

ed etichettare le definizioni con i rispettivi istanti di validità per l'uso futuro. Lo *Schema Evolution*, invece, non richiede la possibilità di etichettare versioni degli schemi, fatta eccezione per il caso in cui ogni schema modificato venga considerato una nuova versione. In aggiunta non richiede che il DBMS fornisca un meccanismo per la visualizzazione delle passate definizioni degli schemi.

Non tutti i cambiamenti dello schema portano necessariamente ad una nuova versione della history. Tipicamente i cambiamenti dello schema saranno più frequenti delle versioni che verranno definite. Le nuove versioni verranno etichettate solo in istanti determinati, quando lo schema ha raggiunto una stabilità, oppure prima di altre modifiche sostanziali che si ritiene non possano fare parte della versione attuale.

Le versioni, inoltre, possono essere etichettate sia nel momento della modifica dello schema (transazione), sia attraverso un metodo definito dall'utente (commit).

Definizione 4 (Schema Version) *Una Schema Version è una versione che compone la History degli schemi del database che viene etichettata dal database administrator.*

Una differenza molto importante tra lo *Schema Evolution* e lo *Schema Versioning* si può notare nell'evoluzione dei domini degli attributi. Nello *Schema Evolution* in presenza di un cambiamento dei domini le istanze esistenti devono obbligatoriamente essere convertite al nuovo formato, e le applicazioni che si basano sulle vecchie strutture devono essere modificate. Nello *Schema Versioning* le definizioni esistenti vengono mantenute, mentre le modifiche vengono apportate solo all'ultima versione.

Lo *Schema Versioning* presenta una serie di problematiche relative all'aggiornamento dei dati attraverso gli schemi storici. La definizione dello schema versioning è quindi rifinita ulteriormente attraverso una distinzione tra le attività di recupero dei dati ed aggiornamento:

Definizione 5 (Partial Schema Versioning) *Un DBMS supporta il* Partial Schema Versioning *quando permette l'accesso a tutti i dati, sia in maniera retrospettiva che prospettica, attraverso versioni che fungono da interfacce e che sono definite dall'utente. Gli aggiornamenti dei dati sono permessi solamente su uno schema definito (solitamente quello corrente).*

Definizione 6 (Full Schema Versioning) *Un DBMS supporta il* Full Schema Versioning *quando permette l'accesso a tutti i dati, sia in maniera retrospettiva che prospettica, attraverso versioni che fungono da interfacce e che sono definite dall'utente. Gli aggiornamenti dei dati sono permessi su tutti gli schemi della history corrispondente.*

Il *Full Schema Versioning* può permettere di rendere asincrone le modifiche agli schemi. Seguendo un approccio di questo tipo è possibile vedere sia i vecchi dati sulle nuove definizioni degli schemi sia i nuovi inserimenti attraverso le vecchie definizioni. L'asincronia delle modifiche comporta notevoli vantaggi sul fronte della stabilità delle operazioni nel tempo, evitando una riscrittura delle applicazioni che manipolano e gestiscono questi dati, ma complica la gestione dei depositi e delle query che si appoggiano su schemi multipli.

Questo ci introduce ad un ulteriore livello di distinzione che raffina ulteriormente la scelta di un sistema per la gestione del versioning, presente quando le versioni dei dati intensionali e quelli estensionali vengono gestite attraverso la stessa dimensione temporale.

- ***Gestione Sincrona***. I dati vengono memorizzati, aggiornati e recuperati sempre attraverso la versione dello schema che condivide la stessa validità temporale dei dati, attraverso le dimensioni temporali comuni. La gestione sincrona degli schemi implica il *versioning sincrono*, dove la validità temporale di una versione dello schema include la validità temporale dei dati corrispondenti attraverso le dimensioni temporali comuni.

- ***Gestione Asincrona***. I dati possono essere recuperati ed aggiornati attraverso qualsiasi versione dello schema, la cui validità è indipendente dalla validità dei dati, anche attraverso le dimensioni temporali comuni. La gestione asincrona degli schemi implica il *versioning asincrono*, dove la validità temporale di una versione dello schema e la validità temporale dei dati corrispondenti sono completamente indipendenti.

Nell'approccio che viene introdotto con questo lavoro di tesi, utilizzeremo il *Partial Schema Versioning*, visto che non si è ritenuto necessario permettere all'utente finale di inserire i nuovi dati nelle le versioni precedenti all'ultima.

Per quanto riguarda l'interrogazione e la gestione dei dati in presenza di versioni multiple degli schemi, in letteratura è stato introdotto il linguaggio `TSQL2` [TSQ95], che rappresenta una estensione dell'SQL con il supporto alla gestione temporale. Tramite il `TSQL2` viene permesso all'utente di scegliere la versione dello schema sulla quale lavorare, tramite la selezione dell'intervallo di validità:

```
SET SCHEMA DATE '31-12-2004'
```

E' chiaro che questo approccio semplifica notevolmente il problema, limitandone notevolmente le potenzialità, ed è utilizzabile nella pratica solo quando si utilizza la tecnica del *completed-schema*, sul quale ci soffermeremo in maniera più approfondita in seguito.

Le interrogazioni che permettono di prendere in esame versioni multiple degli schemi contemporaneamente sono state considerate in [DGS95] e [Gra02].

2.3 Gestione dei dati estensionali

Se fino ad ora abbiamo considerato prevalentemente la gestione degli schemi in presenza di versioning, la gestione dei dati estensionali relativi alle versioni è uno degli aspetti più problematici. Le scelte implementative analizzate in letteratura sono diverse, a seconda degli obbiettivi richiesti dall'ambito di utilizzo e delle semplificazioni che si possono imporre alla gestione dello *Schema Versioning*

Possiamo chiarire con un esempio i diversi livelli di intervento richiesti all'amministratore di un database relazionale in presenza di modifiche allo schema, mostrando i tre casi principali di supporto agli aggiornamenti che abbiamo introdotto e le operazioni di migrazione dei dati che vengono richieste.

Nell'esempio 1 viene creato lo schema di una nuova relazione (*snapshot*) SV_1 e vengono inserite nel catalogo le corrispondenti informazioni.

Esempio 1 *La tabella `IMPIEGATI` è inizialmente vuota, poi vengono inserite le due tuple nella nuova relazione.*

```
CREATE TABLE IMPIEGATI
(NOME CHAR(10) NOT NULL PRIMARY KEY, INDIRIZZO CHAR(20), CITTA CHAR(10));
```

```
INSERT INTO IMPIEGATI
VALUES('Brown', 'King's road 15', 'London');
INSERT INTO EMPLOYEE
VALUES('Rossi', 'Via Veneto 7', 'Rome');
```

Nella tabella che segue viene mostrato il contenuto della relazione relativa allo schema SV_1, che dopo le operazioni eseguite nell'esempio 1 è diventato:

SV_1 IMPIEGATI(NOME, INDIRIZZO, CITTA)

NOME	INDIRIZZO	CITTA
Brown	King's road 15	London
Rossi	Via Veneto 7	Rome

Consideriamo ora i cambiamenti nello schema:

S_1 `ALTER TABLE IMPIEGATI DROP COLUMN ADDRESS;`

S_2 `ALTER TABLE IMPIEGATI ADD COLUMN TELEFONO CHAR(8);`

ed esaminiamo quello che accade nelle tre tipologie di supporto ai cambiamenti dello schema.

Schema modification

In un sistema che supporta solo i cambiamenti dello schema, l'effetto della S_1 ottiene questo risultato:

SV_2 IMPIEGATI(NOME, CITTA, TELEFONO)

NOME	CITTA	TELEFONO
(empty)		

Le vecchie informazioni sono perse e devono essere reinserite dall'utente, possibilmente all'interno della stessa transazione contenente S_1 ed S_2, come fossero nuove tuple. Nelle applicazioni esistenti attualmente in commercio è l'amministratore che eventualmente sceglie di eliminare il contenuto delle relazioni in presenza di una modifica dello schema, altrimenti le tuple vengono modificate solamente negli attributi che vengono modificati.

S_3 `INSERT INTO IMPIEGATI VALUES (BROWN,LONDON, 224466);`

S_4 `INSERT INTO IMPIEGATI VALUES (ROSSI,ROMA, 775533);`

Dopo queste operazioni lo stato finale sarà il seguente:

SV_2 `IMPIEGATI(NOME, CITTA, TELEFONO)`

NOME	CITTA	TELEFONO
Brown	London	224466
Rossi	Roma	775533

Schema evolution

Se il nostro sistema supporta l'evoluzione dello schema, l'effetto dei cambiamenti dell'esecuzione della S_1 possono essere rappresentati dalla tabella che segue:

SV_2 `IMPIEGATI(NOME, CITTA, TELEFONO)`

NOME	CITTA	TELEFONO
Brown	London	*null*
Rossi	Roma	*null*

In questo caso, il contenuto delle vecchie colonne NOME e CITTA sono automaticamente mantenute anche dopo il cambiamento dello schema. Le nuove informazioni possono essere inserite con i seguenti comandi SQL:

S_3 `UPDATE IMPIEGATI SET TELEFONO='224466' WHERE NOME='Brown';`

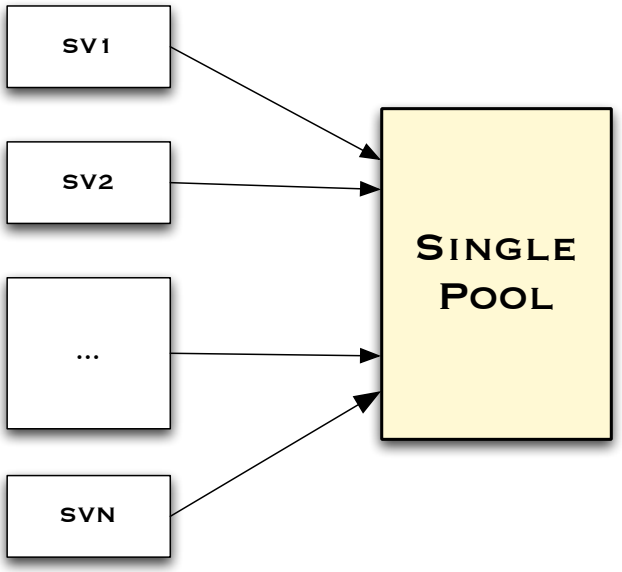
S_3 `UPDATE IMPIEGATI SET TELEFONO='775533' WHERE NOME='Rossi';`

Lo stato finale è lo stesso dell'esempio precedente.

Schema versioning

Se il nostro schema supporta invece lo schema versioning, assumiamo che la tabella IMPIEGATI sia stata creata in data 1/1/1990 ed il cambiamento nello schema S_1 sia stato effettuato nel 7/1/1994. A questo punto, possiamo rappresentare gli effetti del cambiamento dello schema, che sono le due versioni risultati assieme alla loro vista sui dati della relazione IMPIEGATI:

Figura 2.1: Sistema di gestione dei dati estensionali Single-pool



SV_1 `[1/9/1990 - 6/30/94] IMPIEGATI(NOME, INDIRIZZO, CITTA)`

NOME	INDIRIZZO	CITTA
Brown	King's road 15	London
Rossi	Via Veneto 7	Rome

SV_2 `[7/1/94 - ∞] IMPIEGATI(NOME, CITTA, TELEFONO)`

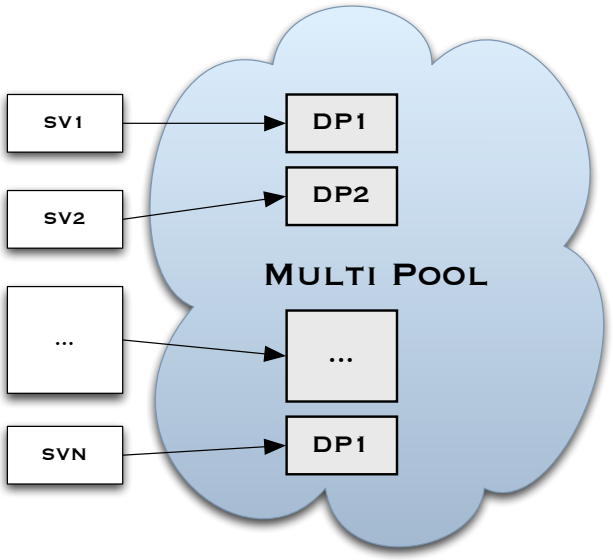
NOME	CITTA	TELEFONO
Brown	London	224466
Rossi	Roma	775533

Dal punto di vista implementativo, nello *Schema Versioning* esistono due diversi tipi di approccio per la gestione ed il mantenimento delle versioni a livello estensionale ([TSQ95]): *single-Pool* e *multi-pool*.

Definizione 7 (Single Pool) *Nella soluzione* single pool *(figura 2.1)tutte le versioni degli schemi sono associate ad un unico e condiviso deposito delle informazioni estensionali, in modo tale che gli stessi oggetti non possono avere differenti valori per le stesse proprietà quando vengono viste attraverso differenti versioni dello schema.*

Definizione 8 (Multi Pool) *Nella soluzione* multi-pool *(figura 2.2) ogni versione di uno schema è associata ad una sorgente privata dei dati estensionali; sorgenti*

Figura 2.2: Sistema di gestione dei dati estensionali Multi-pool



diverse possono contenere gli stessi oggetti ed eventualmente avere rappresentazioni ed evoluzioni completamente differenti.

Sebbene la soluzione *multi-pool* risulti più flessibile, una soluzione *single-pool* viene solitamente considerata soddisfacente per l'implementazione, per esempio nel caso della tecnica del *completed schema*, presentato in [RS95, CGS97].

Secondo l'approccio definito nel *completed schema*, tutte le relazioni sono definite attraverso l'unione di tutti gli attributi che vengono definiti su di loro, incluse quelle che successivamente vengono cancellate. Il `DROP` delle colonne non elimina fisicamente i dati, ma li disattiva in una determinata versione. Il linguaggio `TSQL2` permette di visualizzare il contenuto di tutta la relazione associata al suo *completed schema*, attraverso l'uso dei doppi asterischi nel comando `SELECT`:

```
SELECT ** FROM IMPIEGATI
```

Seguendo l'esempio della tabella degli impiegati, il risultato della query precedente risulterebbe essere:

NOME	INDIRIZZO	CITTA	TELEFONO
Brown	King's Road 15	London	224466
Rossi	Via Veneto 7	Rome	775533

In questo modo, non è necessario effettuare nessuna modifica al modello dei dati. Di fatto, questo modello di *Schema Versioning* non introduce nessun livello di versioning dei dati e può essere basato sul meccanismo classico delle viste. In questo caso, però, non esistendo due diverse versioni delle tuple relative agli stessi impiegati, viene impedita la possibilità di avere due città diverse nelle due versioni. Questo chiarisce ulteriormente il significato delle diverse dimensioni temporali: sebbene lo schema SV_1 sia relativo allo spazio temporale `[1/9/1990 - 6/30/94]`, i dati relativi alle tuple possono essere aggiornati anche successivamente. Se nel 1995 verrà modificata la città di residenza di un impiegato, il dato verrà aggiornato anche nelle relazioni corrispondenti ad SV_1 nonostante lo schema non sia più attuale, e la nuova città sarà disponibile anche per le applicazioni che si basano su SV_1.

2.4 Migrazione dei dati

Data una versione S_1, ed una serie di operazioni di modifica M che portano ad una seconda versione S_2, le operazioni di migrazione dei dati necessarie per popolare S_2 dipenderanno da tutte le tipologie di classificazione che sono state presentate, nei diversi livelli della gestione dello schema versioning.

Data l'estrema varietà delle soluzioni implementative possibili, analizzeremo solo i casi più importanti e le relative operazioni di migrazione.

- ***Gestione Sincrona, single-pool***. Nel caso si utilizzi una soluzione single-pool con una gestione sincrona, attraverso il *completed schema*, tutti i dati estensionali sono memorizzati insieme su un unico deposito. Il *completed schema* contiene tutti gli attributi che sono stati incrementalmente definiti dai successivi cambiamenti. L'introduzione di nuovi attributi nello schema produrrà una lista di valori *null*, che non essendo presenti nelle precedenti versioni non possono essere riempiti. Nel caso un attributo cambi dominio, si dovrà duplicarlo ed associare ognuno dei due a differenti versioni dello schema. Dal punto di vista pratico, questa soluzione presenta il minor numero di operazioni di migrazione possibile, perché ad ogni modifica vengono aggiunti attributi alle relazioni già esistenti e non vengono duplicati quelli presenti in precedenza.

- ***Gestione Sincrona, multi-pool***. Nella gestione sincrona in multi-pool viene gestito un deposito per i dati per ciascuna versione dello schema che viene introdotta. In questo caso, quindi, S_2 dovrà essere popolato di tutti i dati che sono validi nell'intervallo di tempo T_2 $[t_2, \infty]$, che è uguale a quello della versione S_2. I dati presenti in S_1 che non condividono la stessa validità temporale di S_2 non dovranno essere migrati nel nuovo deposito, grazie alla scelta della gestione sincrona.

- ***Gestione Asincrona, multi-pool***. In questo caso, sebbene ogni versione abbia il suo deposito, è necessario migrare e duplicare tutti i dati preesistenti, affinché sia possibile modificarli e recuperarli attraverso la nuova versione.

Capitolo 3

Il problema del tempo nei data warehouse

I data warehouse e le applicazioni che vi si appoggiano rappresentano un potente framework per l'immagazzinamento e l'analisi di enormi quantità di dati finalizzati al supporto alle decisioni. In questo contesto, le analisi effettuate nei data warehouse si focalizzano su insiemi di dati che sono stati raccolti in lunghi periodi di tempo. Un Data warehouse può essere considerato, quindi, come un database specializzato con caratteristiche di raccolta di informazioni temporali o addirittura storiche.

Un cubo di un data warehouse modella un insieme di *eventi*, ed è composto dal *fatto di interesse*, dalla descrizione delle *dimensioni* e dalle *misure*.

Il fatto d'interesse rappresenta gli elementi dell'informazione atomica nel database multidimensionale, ed è il centro concettuale dell'informazione che si vuole mantenere nel data warehouse. Nel caso dei data warehouse memorizzati in un database relazionale, ad esempio, i dati relativi al fatto vengono memorizzati nella *fact table* e rappresentano gli eventi che sono al centro dello studio del cubo multidimensionale. Le dimensioni rappresentano il contesto corrispondente al fatto e sono strutturate secondo una gerarchia di dipendenze funzionali che ne descrivono i vari livelli di aggregazione, le *proprietà*. Le misure sono i valori quantitativi immagazzinati nel data warehouse relativi al fatto d'interesse e sono organizzate eventualmente in una gerarchia che ne descrive le funzioni di derivazione, nel caso delle *misure derivate*.

Allo stato attuale esistono diversi formalismi che permettono di descrivere gli

schemi dei cubi, gli attributi e le gerarchie delle dipendenze funzionali, ognuno dei quali ha caratteristiche e funzionalità proprie. Negli schemi concettuali che descrivono un data warehouse, le dimensioni sono grafi orientati (ciclici o aciclici, a seconda dei formalismi), i cui nodi rappresentano le proprietà corrispondenti ai livelli di aggregazione di interesse, ed i cui archi rappresentano le dipendenze funzionali.

I dati inseriti all'interno di un data warehouse attraverso le diverse tipologie di fonti di alimentazione, quindi, sono sempre visti in un contesto temporale. Ciò avviene a causa della peculiarità dell'immagazzinamento delle informazioni relative al fatto di interesse, che avviene in maniera incrementale nel tempo. Oltre a questo, si deve tenere in considerazione la possibilità che anche le stesse dimensioni e misure possano subire cambiamenti durante questo lungo periodo di tempo.

Le modifiche alle fonti di alimentazione di un data warehouse, quindi, possono essere suddivise in due categorie distinte: quelle che coinvolgono cambiamenti nei dati (inserimento, aggiornamento, cancellazione di tuple) e quelle che corrispondono a cambiamenti nella struttura (schema) che li modella. Entrambe queste categorie possono portare a modifiche dello schema del data warehouse.

Come soluzione al problema della gestione delle modifiche alle fonti di alimentazione si potrebbe decidere di isolarle attraverso uno strato intermedio di middleware, che riconduca gli elementi di informazione alla struttura inizialmente adottata per l'alimentazione del data warehouse. Un approccio di questo tipo non può però protrarsi nel tempo, perché porta ad inconsistenze dovute all'allontanamento incrementale tra la realtà modellata a livello delle fonti e quella gestita dal data warehouse.

Un metodo molto più serio ed inevitabilmente più complesso da gestire, invece, deve prevedere una propagazione dei cambiamenti ad ogni livello, tenendo traccia delle modifiche a livello dell'alimentazione ed applicandole alla definizione del data warehouse. Le due diverse modalità possibili di gestione di questa propagazione sono simili a quelle individuate nei database relazionali: evoluzione e versioning.

Come esempio possiamo pensare ad un data warehouse che studia le caratteristiche politiche ed economiche dei paesi del continente europeo degli ultimi venti anni. E' chiaro che qualsiasi analisi di questo tipo non potrebbe ignorare i cambiamenti politici che hanno modificato la mappa dell'Europa in questo periodo di tempo. L'Unione Sovietica, La Germania, la Cecoslovacchia, etc. hanno subito

profondi cambiamenti politici che ne hanno modificato persino i confini geografici. Nelle interrogazioni che tengono in considerazione diversi periodi di tempo sia precedenti sia successivi a questi cambiamenti, il sistema deve gestire il problema della comparabilità dei dati e delle conseguenze di queste differenze sui risultati.

Tramite una evoluzione degli schemi, in questo caso sarebbe possibile adattare il sistema all'attualità, ma si perderebbe la possibilità di interrogare il data warehouse attraverso gli schemi precedenti ed i dati corrispondenti. Lo schema versioning, nelle sue varie forme, permette di gestire questa situazione e di mantenere uno storico delle modifiche apportate agli schemi.

Sorprendentemente, la ricerca non ha ancora sufficientemente approfondito questo ambito, e nessuna soluzione fino ad ora presentata risulta essere completa.

Uno dei motivi per i quali la gestione temporale dei cambiamenti nei data warehouse non è ancora stata approfondita è la considerazione che sta alla base del loro sviluppo, che prevede che le dimensioni siano ortogonali tra loro. In presenza di una dimensione temporale, come accade nella quasi totalità delle implementazioni, l'assunzione dell'ortogonalità delle dimensioni porta alla necessità che queste siano invarianti nel tempo. Questa implicita assunzione, ovviamente, è molto limitante e poco realistica.

L'analisi delle problematiche relative alla gestione del tempo nei data warehouse può essere quindi suddivisa in due sezioni, riguardanti rispettivamente gli aspetti *estensionali* ed *intensionali* della gestione del tempo nei data warehouse, anche se è ovvio che profondi cambiamenti apportati ad un livello si propaghino anche nell'altro.

3.1 Livello estensionale

Nei data warehouse l'evoluzione nel tempo degli schemi coinvolge dimensioni e misure. Il fatto d'interesse non è solitamente soggetto a variazioni, dato che il cubo multidimensionale si basa sulla sua centralità e sull'importanza che questo ricopre all'interno del dominio che si sta studiando.

Uno dei nodi centrali della gestione dell'evoluzione del cubo multidimensionale

Tabella 3.1: Esempio di relazione temporale R

id	dept	name	office	phone	T
123456	Research	Amy	121	1-2345	[01/01/91.05/01/92].[01/01/94.06/01/96]
123456	Research	Amy	151	1-5432	[06/01/96.12/31/96]
700000	Development	Ben	B07	7-0000	[11/01/96.05/01/97]
714285	Development	Coy	B17	7-1428	[11/01/96.NOW]

è lo studio dei cambiamenti nelle dimensioni. In letteratura si utilizza il termine *Changing Dimensions* quando le relazioni delle dimensioni vengono aggiornate nel tempo, modificando alcune tuple già inserite. Quando la percentuale delle tuple modificate rispetto a quelle totali è inferiore al 2%-5%, queste dimensioni sono chiamate *Slowly Changing Dimensions*.

A livello estensionale, nei data warehouse esistono problematiche da gestire molto simili a quelle presenti nei database relazionali. Nel caso particolare dei sistemi ROLAP (Relational OnLine Analytical Processing), dove il backend delle applicazioni si basa su database relazionali, le soluzioni per l'immagazzinamento delle informazioni relative all'evoluzione o al versioning degli schemi sono quelle utilizzate nei database temporali. Per questo motivo gli approcci di evoluzione e versioning sfruttano la ricerca effettuata in precedenza in questo campo, introducendo innovazioni principalmente nella modellazione e gestione degli schemi di data warehousing e nella consistenza dei diversi livelli che compongono i sistemi informativi di questo tipo.

In [Yan01] viene analizzato un metodo per la gestione dell'evoluzione delle dimensioni, organizzate in viste materializzate che supportano una dimensione temporale. L'immagazzinamento delle relative informazioni temporali avviene inserendo un apposito attributo *timestamp* nel database relazionale di supporto, sul quale viene inserito il modello temporale introdotto e sul quale vengono tradotte le query temporali.

Nella tabella 3.1 viene mostrato un esempio di relazione temporale. L'attributo T definisce la validità temporale della tupla di riferimento, come composizione di periodi temporali composti da *istanti* (*chronon*), sui quali viene definito un ordinamento totale $<$.

In questo modello non viene analizzata la possibilità di effettuare query multi-schema, ma viene permesso il mantenimento e l'interrogazione in uno specifico periodo temporale.

L'evoluzione degli schemi introduce inoltre problematiche di consistenza e di qualità dei dati presenti nel data warehouse. Diversi studi in letteratura hanno finora affrontato l'impatto dell'evoluzione sulla qualità dei processi di mantenimento del data warehouse e sulla consistenza dei dati, senza analizzare il versioning degli schemi e conseguentemente la possibilità di effettuare interrogazioni che coinvolgono versioni diverse.

In [Qui99] viene analizzato e discusso in termini generali l'impatto dell'evoluzione sulla qualità dei processi di data warehousing, proponendo come soluzione un meta-modello di supporto. In questo studio vengono prese in esame le necessità di modifiche al data warehouse sotto tre diverse prospettive:

- **A livello concettuale**, che rappresenta una prospettiva di business complessiva sulle risorse informative e sulle operazioni di analisi di una azienda.
- **A livello logico**, descrivendo gli schemi utilizzati nelle sorgenti, nel data warehouse, e nei data mart.
- **A livello fisico**, che corrisponde alla gestione dei metodi con i quali i dati vengono fisicamente memorizzati.

Ciascuna di queste prospettive ha tre differenti livelli: il sorgente, il data warehouse e quello del cliente. Un ruolo centrale in questo modello viene giocato dal modello del data warehouse, che dovrebbe essere una rappresentazione concettuale dei dati che sono disponibili nell'azienda.

Il meta-modello presentato attraverso il progetto Data Warehouse Quality (DWQ) [JJQV99] per la gestione dei meta-dati in un data warehouse tiene traccia sia delle componenti architetturali sia dei fattori qualitativi che lo caratterizzano.

La figura 3.1 mostra il framework 3x3 sul quale si basa il modello, che identifica le tre prospettive (*concettuale, logica e fisica*) ed i tre livelli (*sorgente, data warehouse, cliente*). Il primo strato della figura 3.1 corrisponde al meta-modello proposto e fornisce una notazione per le entità generiche per il data warehouse, come lo schema,

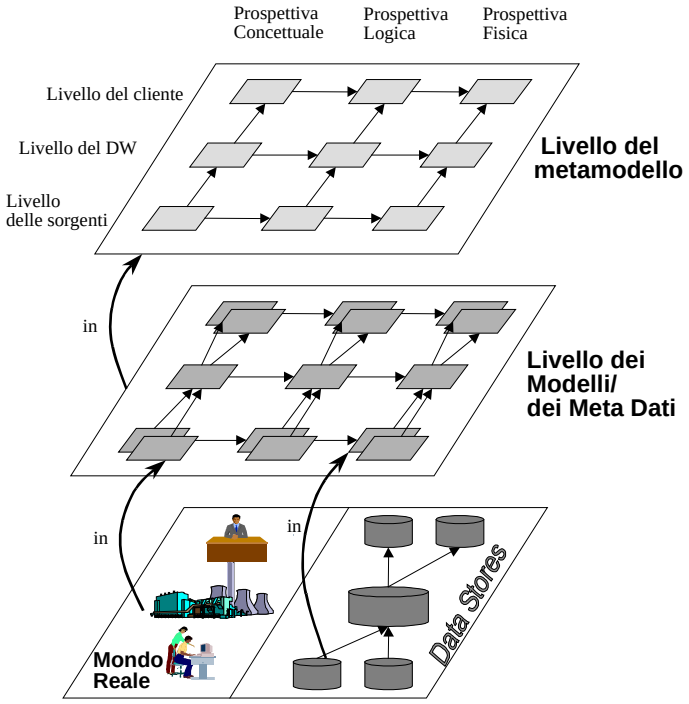


Figura 3.1: Meta-modello DWQ dell'architettura del data warehouse

inclusa la prospettiva di business. Il meta-modello viene istanziato con i meta-dati del data warehouse, che sono mostrati nel secondo strato della figura 3.1, come le definizioni degli schemi relazionali o la descrizione del modello concettuale del data warehouse. Il terzo strato rappresenta il mondo reale nel quale i dati attuali risiedono: qui i meta-dati sono istanziati con le istanze dei dati, come ad esempio le tuple di una relazione o gli oggetti del mondo reale che sono rappresentati dalle entità del modello concettuale.

Ogni oggetto presente nei tre livelli e nelle tre prospettive del framework può essere soggetto a misure di qualità, che sono inserite direttamente nella modellazione. Questo significa che il modello per la qualità è parte integrante del deposito dei meta-dati, e le informazioni riguardanti la qualità sono esplicitamente connesse agli oggetti architetturali.

Nella figura 3.2 viene mostrato il modello per la qualità DWQ. La classe `ObjectType` si riferisce ad un qualsiasi meta-oggetto del framework DWQ, mostrato nel primo strato della figura 3.1. Un *obbiettivo di qualità* (`quality goal`) è una necessità

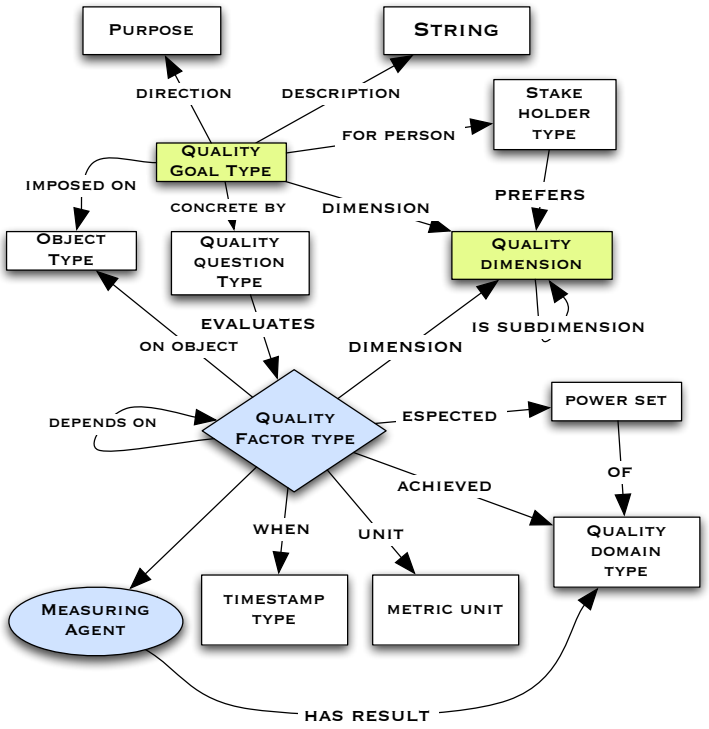


Figura 3.2: Modello di qualità DWQ

astratta di interesse per lo stakeholder, definita su un tipo di oggetto. Un obbiettivo di qualità esprime in maniera diretta alcune richieste in un linguaggio naturale, come ad esempio la disponibilità della sorgente S_1 fino alla fine del mese per l'amministratore del database. Le *dimensioni di qualità* (`quality dimensions` (ad esempio disponibilità) sono utilizzate per classificare gli obbiettivi in categorie differenti. Un *fattore di qualità* (`quality factor`) rappresenta una misura attuale del valore della qualità, e connette quindi i valori di qualità ad oggetti misurabili. La misura di questi fattori di qualità avviene attraverso gli agenti di misura (`measuring agents`).

Il workflow necessario per la costruzione ed il mantenimento del data warehouse viene descritto attraverso un modello per i processi (figura 3.3), che raccoglie le problematiche principali della gestione delle operazioni sul data warehouse. In questo modello viene introdotta la gestione delle evoluzioni del data warehouse, ed analizzato il suo impatto sulla qualità dei processi.

Per un maggiore controllo dell'evoluzione vengono introdotti dei meta-dati che tengono traccia della storia dei cambiamenti e forniscono un insieme di regole di consistenza da rafforzare quando un fattore di qualità deve essere rivalutato. Il

Tabella 3.2: Operatori di evoluzione per le relazioni di base e le viste ed i loro effetti sulla qualità dei data warehouse

Operatori di evoluzione	Effetti sui fattori di qualità	Livello sul quale operano
Aggiunta di una relazione di base/vista	- Completezza, correttezza e consistenza dello schema logico e del modello concettuale - Utilità dello schema - Disponibilità del data store	- Relazione - Logical Schema - Data store
Cancellazione di una relazione di base/vista	- Minimalità dello schema logico - Completezza, correttezza e consistenza dello schema logico e del modello concettuale - Disponibilità del data store	- Relazione, Schema Logico - Data Store - Vista - Agente per il mantenimento delle viste
Aggiunta di un attributo alla relazione di base/vista	- Completezza, correttezza e consistenza dello schema logico e del modello concettuale - Interpretabilità della relazione - Ridondanza degli attributi	- Relazione - Data store - Vista - Agente per il mantenimento delle viste
Cancellazione di un attributo dalla relazione di base/vista	- Completezza, correttezza e consistenza dello schema logico e del modello concettuale - Interpretabilità della relazione - Ridondanza degli attributi	- Relazione - Data store - Vista - Agente per il mantenimento delle viste
Cambiamento del nome di una relazione, vista o attributo	- Interpretabilità e comprensibilità della relazione e dei loro attributi	- Relazione, vista - Data Store, Agente VM
Modifica del dominio di un attributo	- Interpretabilità dei dati	- Relazione, vista - Data Store, Agente VM
Aggiunta di un vincolo di integrità	- Credibilità e consistenza dei dati nel data store	- Schema Logico - Data Store
Cancellazione di un vincolo di integrità	- Consistenza dei dati e dei vincoli di integrità	- Schema logico - Data store
Cambiamento nella definizione di una vista	- Completezza, correttezza e consistenza dello schema logico e del modello concettuale - Utilità dello schema	- Vista - Data Store - Agente per il mantenimento delle viste

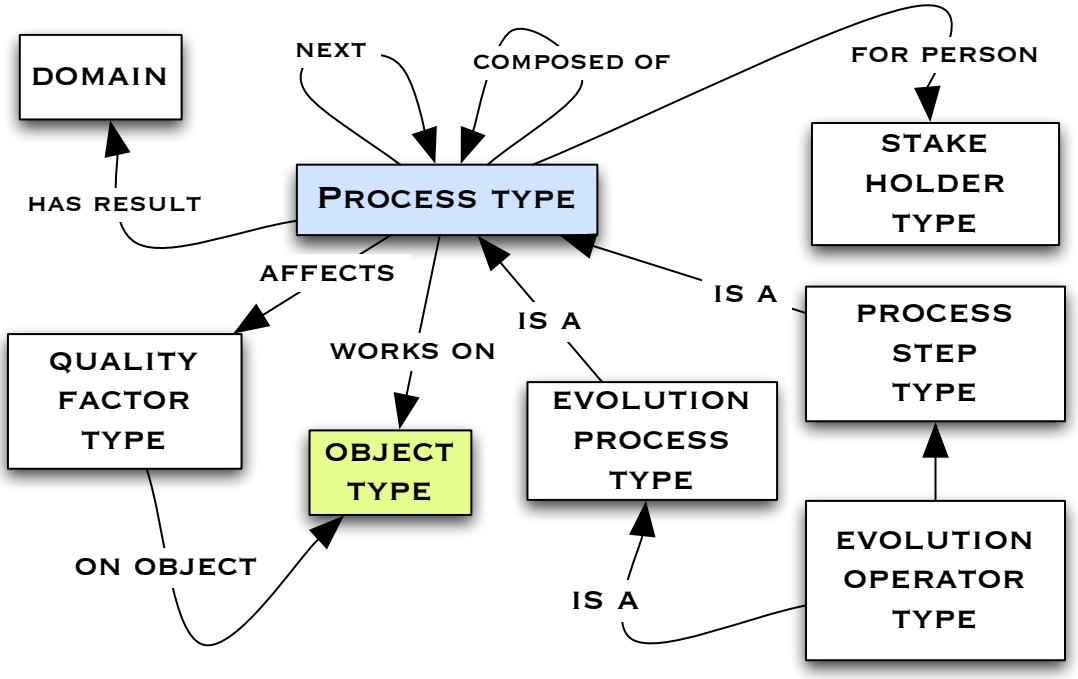


Figura 3.3: Modello di processo di data warehousing specializzato nella evoluzione

processo di evoluzione è composto da operatori di evoluzione e da processi standard. Un tipico processo di evoluzione è ad esempio la materializzazione della vista di un nuovo data warehouse (figura 3.4). Questo processo include, tra le altre, le operazioni di aggiunta di una nuova relazione allo schema del data warehouse e quelle di inserimento, estrazione e scrittura necessarie a valutare una vista e registrarne il contenuto.

Per gestire il controllo di qualità viene tenuta traccia dei processi di evoluzione che hanno modificato la configurazione ed i dati del data warehouse. Attraverso queste informazioni, per ogni eventuale problema di qualità è possibile stabilire quale processo lo ha causato.

Nella tabella 3.2 vengono elencati gli operatori di evoluzione per le relazioni di base e le viste, collegati ai fattori di qualità che vengono coinvolti dalla loro applicazione.

In [BEK+04] viene presentato un modello di data warehouse multidimensionale basato sul versioning, introducendo il concetto di versioni alternative. Accanto alle *versioni reali*, basate sui cambiamenti del dominio applicativo, le versioni alternative

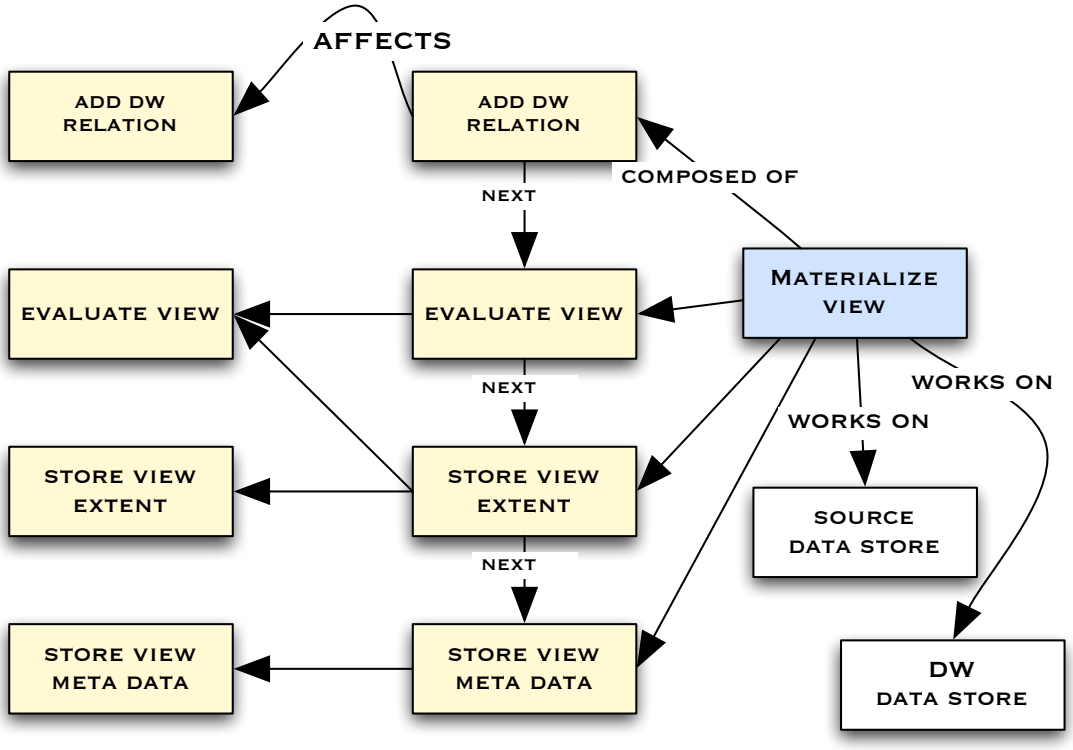


Figura 3.4: Evoluzione del data warehouse: materializzazione di una vista

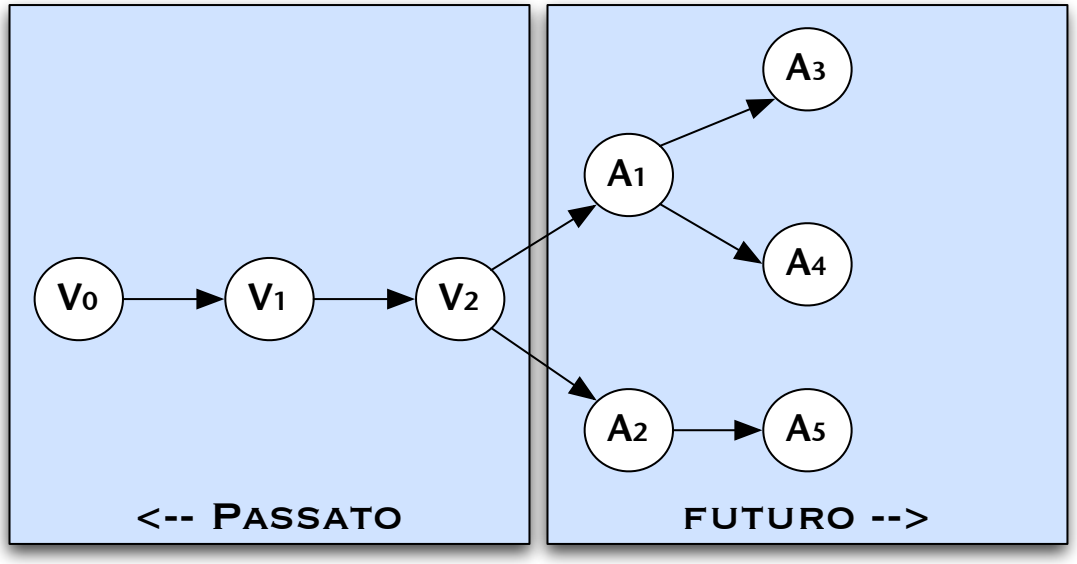


Figura 3.5: Grafo di derivazione delle versioni

permettono di confrontare le analisi in risposta ad interrogazioni del tipo *what-if*, improntate su ipotesi applicate ad eventuali futuri cambiamenti del dominio. Attraverso queste versioni l'utente riesce a ricavare delle proiezioni dei risultati basate su ipotesi di cambiamento, ed introdurre decisioni applicate direttamente al dominio in base a questi scenari alternativi. Questo modello multidimensionale è formato da un grafo di derivazione delle versioni (reali ed alternative), ognuna derivata esplicitamente da una versione precedente. Ad ogni versione viene quindi associato un insieme di dati consistenti corrispondenti. Anche in questo caso ogni versione è valida per un determinato periodo di tempo, individuato da un istante iniziale ed uno finale, ma più versioni alternative possono condividere lo stesso periodo di validità. Come è possibile vedere nella figura 3.5, il grafo di derivazione permette al progettista di gestire le versioni reali ed alternative sia in fase di modellazione che di interrogazione. $\{V_1, V_2, V_3\}$ è l'insieme delle versioni reali, $\{A_1, A_2, A_3, A_4, A_5\}$ è quello delle versioni alternative. Le versioni che sono sovrapposte in verticale (quindi hanno la stessa ipotetica proiezione sull'asse orizzontale) hanno la stessa validità temporale.

Tabella 3.3: Operatori di aggiornamento strutturale

Operatori di Aggiornamento	Descrizione
Generalize	Aggiunge un nuovo livello alla dimensione sopra l'attuale
Relate	Collega due livelli indipendenti di una dimensione
Unrelate	Elimina una connessione tra due livelli di una dimensione
Delete Level	Elimina un livello e le sue funzioni di roll-up
Specialize	Aggiunge un livello alla dimensione, sotto quello inferiore

Tabella 3.4: Operatori di aggiornamento delle istanze

Operatori di Aggiornamento	Descrizione
Add Instance	Aggiunge un elemento in un livello della dimensione
Del Instance	Cancella un elemento in un livello della dimensione

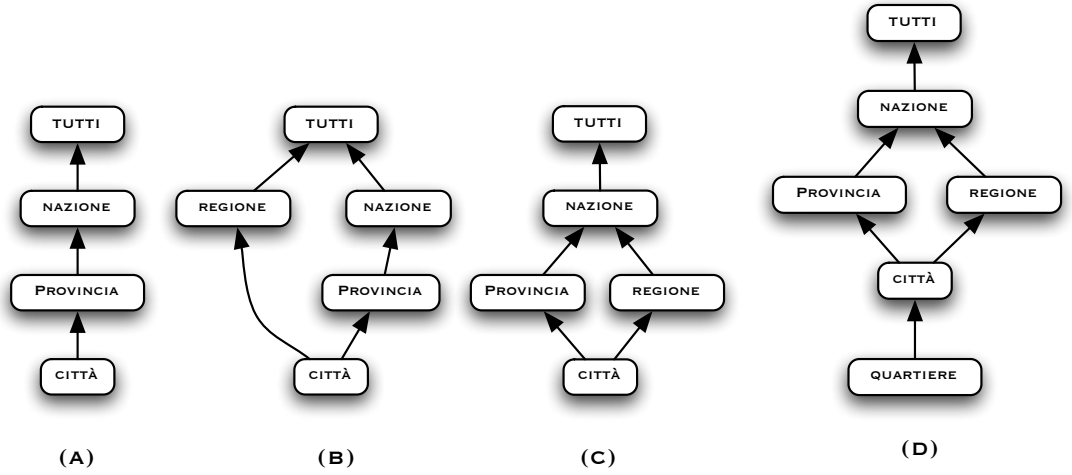
3.2 Livello intensionale

In [VMRC02] viene presentato un prototipo che supporta gli aggiornamenti delle dimensioni, sia a livello degli schemi che su quello dei dati.

Entrambi questi differenti livelli hanno un insieme di operatori di aggiornamento: a livello degli schemi insistono gli operatori strutturali, descritti in dettaglio nella tabella 3.3 , mentre sul livello dei dati lavorano gli operatori delle istanze, elencati nella tabella 3.2. Molte delle operazioni di modifica che lavorano sui dati sono il frutto della composizione di queste operazioni elementari, sulle quali possono venire definite le operazioni complesse di modifica delle istanze.

Esempio 2 *La figura 3.6 mostra un esempio di una dimensione che descrive una localizzazione geografica, aggiornandola attraverso una sequenza di operazioni strut-*

Figura 3.6: Esempio di aggiornamento di una dimensione



turali. Rispetto al primo grafo (a), il secondo (b) mostra una generalizzazione del livello della città al livello della regione, il terzo (c) mostra lo schema della dimensione dopo aver collegato il livello regione con il paese, ed infine il quarto mostra una specializzazione del livello della città ad una granularità più fine, il quartiere.

Questo esempio chiarisce anche i limiti di un approccio basato sull'evoluzione: le istanze che corrispondono al grafo (a) della figura 3.6 devono essere tutte adattate alla nuova descrizione (d) introdotta nelle varie fasi successive. Non è quindi possibile fare riferimento alle vecchie descrizioni attraverso i gli schemi che sono stati sostituiti attraverso le nuove modifiche.

In [LHV02] viene studiato il problema della consistenza di dati relativi alle dimensioni in presenza di una evoluzione nella loro gerarchia.

Definizione 9 (Schema delle dimensioni) *Uno* schema di dimensione *è un grafo orientato aciclico $D_S = (L_S, F_S)$ dove L_S è un insieme fissato di livelli o nodi ed F_S è un insieme fissato di archi orientati tra i nodi.*

Il grafo ha le seguenti proprietà:

1. Definiamo come grado interno di un nodo il numero dei livelli che lo precedono a partire dalla radice e come grado esterno il numero dei livelli di un nodo a partire dall'ultimo livello. Esiste un distinto nodo terminale $l_t \in L_S$ di livello

interno 0 ed un implicito nodo $l_{all} \in L_S$ di grado esterno 0. Il grado interno ed esterno di tutti gli altri nodi in L_S è più grande di zero.

2. Se un livello $l_i \in L_S$ determina funzionalmente il livello $l_j \in L_S$ (cioè se $l_i \rightarrow l_j$ nella notazione standard dei database), che significa che $(l_i, l_j) \in F_S$, allora non c'è un altro percorso da l_i ad l_j attraverso uno o più nodi intermedi.

3. A ciascun livello delle dimensioni è associato un valore di dominio. In particolare, il valore del dominio del livello l_{all} è $\{all\}$.

Una dimensione presente in più di uno schemi delle dimensioni è chiamata *dimensione multipla*.

Una dimensione multipla $D_1 = (L_1, F_1)$ su più schemi di dimensione $D_S = (L_S, F_S)$ è consistente se sono soddisfatte le seguenti condizioni:

1. Se c'è più di un percorso nello schema dal livello l_i al livello l_j, dove $l_i, l_j \in L_S$, allora i percorsi differenti da un membro di l_i ad un membro di l_j devono portare allo stesso elemento di l_j.

2. Se c'è un solo percorso dallo schema l_i allo schema l_j per $l_i, l_j \in L_S$, allora c'è anche solo un percorso per qualsiasi membro di l_i ad un membro di l_j.

Si nota, quindi, che una dimensione su uno schema che ha solo un percorso da un livello l_t ad l_{all} è sempre consistente.

Dato uno schema di dimensione D_S definito come sopra, $D_S^+ = (L_S, F_S^+)$ denota la chiusura transitiva di D_S.

L'insieme dei diretti successori di un dato livello $l_i \in L_S$ è definito come $Succ_{l_i} = \{l_x | (l_i, l_x) \in F_S\}$, mentre l'insieme di tutti i successori di l_i è definito come $Succ_{l_i}^+ = \{l_x | (l_i, l_x) \in F_S^+\}$

Per controllare possibili conflittualità durante gli inserimenti di nuovi livelli in uno schema di dimensione viene quindi introdotto il concetto di livelli di conflitto, definito come segue.

Definizione 10 (Livelli di conflitto) *Sia dato uno schema di definizione $D_S = (L_S, F_S)$ con la chiusura transitiva $D_S^+ = L_S, F_S^+$. Un livello $l_j \in L_S$ è un* livello di conflitto *per una operazione di inserimento nel livello $l_i \in L_S$ se sono soddisfatte le seguenti condizioni:*

1. $outdeg(l_i) = |Succ_{l_i}| > 1$

2. $\exists l_x, l_y \in Succ_{l_i}, l_x \neq l_y : l_j \in Succ^+_{l_x} \cap Succ^+_{l_y}$

Intuitivamente la condizione 1 richiede che il livello l_j sia direttamente connesso a più di un livello superiore della gerarchia delle dimensioni, mentre la condizione 2 afferma che un conflitto di inserimento può occorrere al livello l_j se il livello l_j può essere raggiunto da almeno due diretti livelli successivi di l_i.

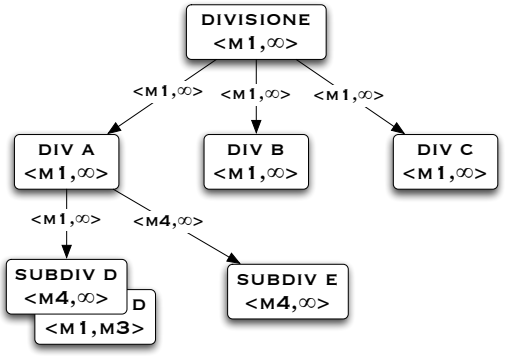
Quando si inserisce un livello nello schema di dimensione, si ha un conflitto in una istanza di dimensione se lo schema ha livelli di conflitto per l'inserimento. In questo caso, deve essere eseguito un test per gli inserimenti nell'istanza della dimensione. Questi conflitti, in ogni caso, avvengono solo se le consistenti dimensioni iniziali subiscono qualche cambiamento. Per questo motivo, le dimensioni potrebbero essere suddivise in due categorie, *changing* and *stable*, a seconda che possano o non possano mutare nel tempo. In alcuni casi la storia dei cambiamenti deve essere mantenuta. Alcuni modelli utilizzano dei timestamp *valid-time* per registrare il periodo di validità delle dipendenze funzionali nello schema e nel grafo dell'istanza, per avere la possibilità di ricostruire lo schema della dimensione e quello dell'istanza in un preciso momento.

Quando un data warehouse è in uso, la conoscenza dell'esistenza di livelli di conflitto all'interno di uno schema di dimensione può essere quindi utilizzato come aiuto per il mantenimento della consistenza dei dati della dimensione. Ogni volta che una dimensione con un livello di conflitto viene aggiornata, può essere testata automaticamente per controllare il verificarsi di un conflitto. Questa possibilità è particolarmente rilevante quando una parte dei dati dimensionali non sono importati da sistemi automatici ma sono mantenuti manualmente o in maniera semiautomatica dall'amministratore del database.

In [EK01] vengono definite le *strutture di versione* di un database temporale, simili agli *schemi di dimensione* visti in precedenza.

Definizione 11 (Versione di struttura) *Una versione di struttura è una vista su una struttura multidimensionale che è valida per un intervallo di tempo definito $[T_sT_e]$, dove T_s è l'istante iniziale, e T_e è quello finale.*

Figura 3.7: Una dimensione Divisione con i relativi timestamp



Concettualmente ad ogni versione di struttura SV corrisponde un cubo con la stessa validità temporale. Ad ogni elemento della struttura viene quindi associato un periodo di validità, come mostrato in figura 3.7, ed è quindi possibile ricostruire la struttura relativa ad un preciso periodo temporale.

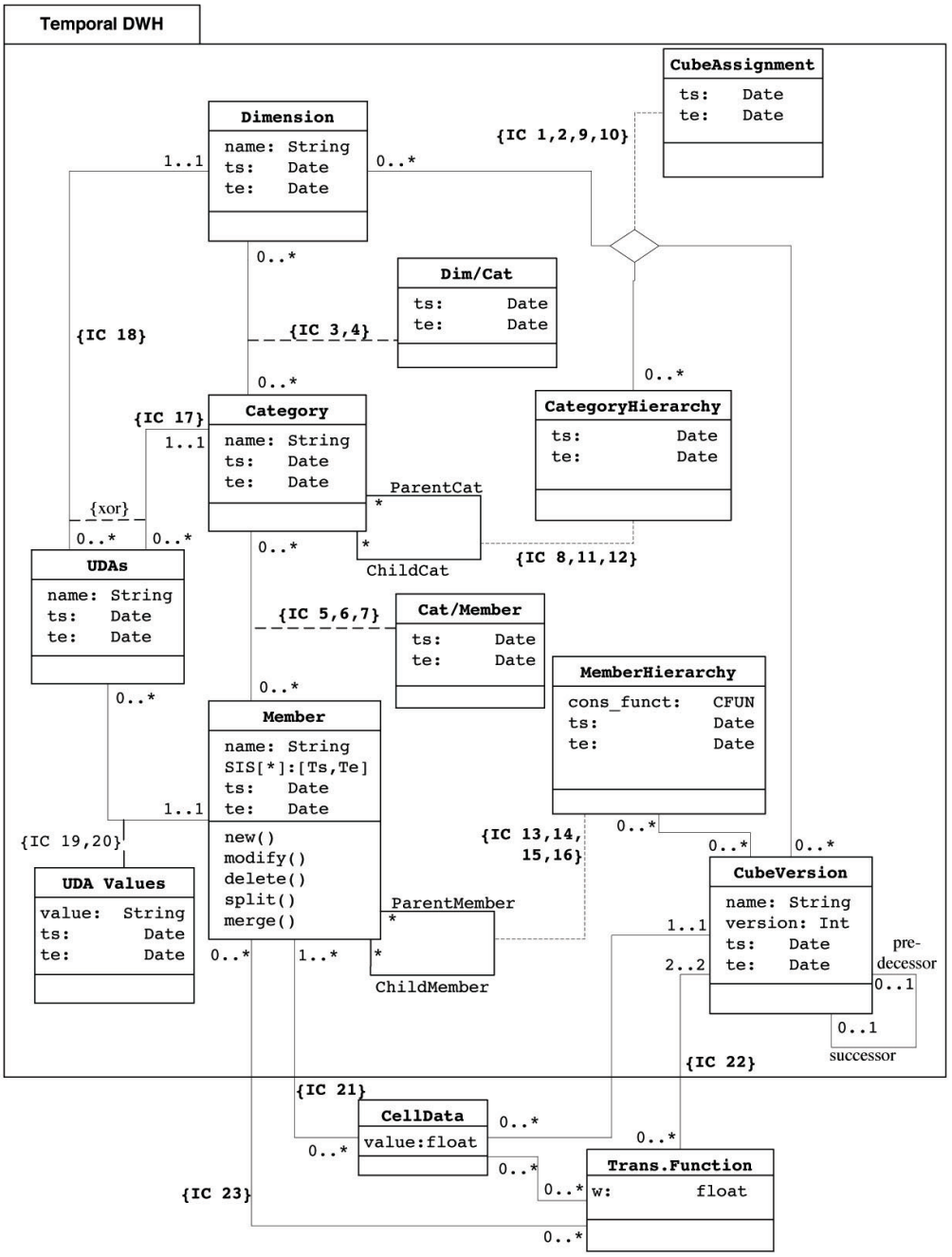
Attraverso la definizione di apposite *matrici di trasformazione* vengono mappate le corrispondenze tra strutture di versioni diverse, in modo tale che dai dati relativi ad una versione si possano ricavare quelli di un'altra.

In questo modello le istanze dei vari livelli di dimensione (proprietà), vengono chiamate membro della dimensione. Ogni membro di una dimensione ha un *timestamp*, un istante di validità, attraverso il quale viene inserito in una versione di struttura.

In [EKM02] viene presentato il modello COMET di supporto all'evoluzione ed il versioning degli schemi. Come si vede nel diagramma UML della figura 3.8, ad ogni oggetto viene associato un intervallo temporale di validità, ed ad ogni cubo viene associata una versione (`cubeVersion`), ad indicare che le diverse istanze del cubo corrispondono a versioni diverse dello schema di riferimento.

Sebbene con l'introduzione del modello COMET venga citato il problema dell'esecuzione delle query che si applicano a `cubeVersion` diversi, le problematiche connesse non sono state esplorate e le modalità con le quali le differenti istanze vengono mappate tra una versione e l'altra non sono state approfondite.

Figura 3.8: Diagramma UML del modello COMET per la gestione dell'evoluzione e del versioning degli schemi di data warehouse



3.3 Interrogazione in presenza di versioning

Quasi tutti gli approcci introdotti in letteratura non permettono l'interrogazione di più schemi differenti. L'introduzione del versioning degli schemi non apporta quindi il vantaggio di poter considerare insiemi consistenti di informazioni relative a versioni differenti. In [EK01] e [Yan01], ad esempio, l'utente deve sempre definire la struttura attraverso la quale desidera interrogare i dati. Nel primo approccio, però, vengono definite delle *matrici di trasformazione* che permettono di ricavare i valori relativi alle dimensioni che cambiano. L'utente, quindi, sceglie una versione indicando l'istante di riferimento P e richiede i dati relativi a periodi temporali anche differenti a quello individuato da P. Il modello individua prima la versione coinvolta, poi trasforma i dati relativi alle altre versioni nella struttura selezionata. Questo modello è disponibile solamente quando si riescono a fare trasformazioni tra un attributo ed un altro modificato della dimensione, cosa che non sempre è possibile. Nonostante questa limitazione, il modello presentato in [EK01] permette quindi di interrogare la base di dati nella sua interezza, in corrispondenza di una unica versione di struttura.

In [EKM02] questa soluzione viene ulteriormente sviluppata, proponendo come soluzione a questo problema il modello COMET, che permette di interrogare versioni differenti. Per ottenere questo risultato viene introdotto il concetto di *intervallo stabile*.

Definizione 12 (Intervallo Stabile) *Un* Intervallo Stabile *per un membro di una dimensione DM è un intervallo nel quale nessuna modifica ad altre componenti influisce sui risultati delle query su DM.*

Come abbiamo già visto, ciascun membro delle dimensioni e le corrispondenti dipendenze funzionali hanno un timestamp che rappresenta il tempo di validità dell'oggetto corrispondente. Questo purtroppo non garantisce che le modifiche ai membri di livello superiore non influiscano sui dati calcolati per il DM che viene preso in considerazione durante l'intervallo di validità dato.

Consideriamo ad esempio un data warehouse temporale che registra le informazioni di una Facoltà. Anche se la Facoltà non viene modificata, i dipartimenti e lo staff al suo interno possono essere cambiati. A questo punto, se una query richiede i dati corrispondenti alla facoltà all'interno dell'intervallo temporale di validità

della stessa, questi potrebbero dipendere da calcoli sbagliati dovuti all'evoluzione strutturale dei dipartimenti.

Gli intervalli di stabilità sono quindi degli insiemi di intervalli temporali dove per ciascun intervallo la struttura che influisce sui dati di questo membro non è cambiata. Per questo motivo, se la query rimane all'interno dell'intervallo di stabilità non c'è nessun necessità di trasformare i dati.

Questa possibilità è una garanzia della consistenza dei risultati delle interrogazioni, ma pone una seria limitazione all'espressività ed alla potenzialità delle query, che non possono estendersi oltre gli intervalli di stabilità.

La possibilità di gestire query multi-schema è strettamente connessa all'individuazione di una struttura di dati comparabile tra gli schemi differenti. Spesso le modifiche apportate allo schema possono essere ignorate per lo scopo dell'interrogazione, oppure possono essere trasportate nelle versioni precedenti per facilitare il compito dell'analisi di uno schema comune. Analizzando per esempio il caso in cui sia stato inserito un nuovo livello di aggregazione nella descrizione dei prodotti di una azienda, aggiungendo una proprietà `sotto-categoria` tra il `prodotto` e la `categoria`. Le due versioni differiscono solamente per questo nuovo livello di aggregazione. L'utente dovrebbe poter scegliere se:

- Aggiornare i dati della versione precedente per inserire le sotto-categorie anche nelle dimensioni e nelle istanze
- Ignorare le sotto-categorie nel caso di interrogazioni che richiedono le istanze di entrambe le versioni, utilizzando solo i dati in comune
- Eseguire l'interrogazione solo su una specifica versione e le relative istanze.

Nel prossimo capitolo verrà quindi introdotto un approccio che tiene conto di queste esigenze, permettendo l'esecuzione di query multi-schema e su schemi singoli, a seconda delle necessità. Questo risultato verrà ottenuto garantendo la possibilità di trasportare nelle vecchie versioni le nuove modifiche apportate senza alterarne la struttura originaria, che rimarrà sempre a disposizione per le interrogazioni su schemi singoli.

Capitolo 4

Un approccio al versioning nei data warehouse

In questo capitolo introdurremo il nostro approccio al problema del versioning degli schemi di data warehousing, basandoci su un esempio di funzionamento. Successivamente formalizzeremo la rappresentazione degli schemi di data warehousing ed introdurremo il concetto di *Schema Aumentato*, il punto chiave del nostro modello.

Consideriamo uno schema S_0 che modella la vendita di componenti (`part`) a clienti sparsi per tutto il mondo. Uno schema concettuale del fatto della vendita (`Shipment`) è mostrato nella figura 4.1(a), utilizzando il formalismo DFM (Dimensional Fact Model) [GMR98].

Il fatto `shipment` mostrato nella figura 4.1(a) ha due misure, rispettivamente `QtyShipped` e `ShippingCosts`, e cinque dimensioni, `Date, Part, Customer, Deal` e `ShipMode`. Una gerarchia di proprietà è connessa ad ogni dimensione. Il significato di ciascun arco è quello dell'associazione *molti-a-uno*, cioè quello di una dipendenza funzionale.

Ora supponiamo che all'istante $t_1 = 1/1/2003$, S_0 subisca una revisione allo scopo di adattarsi meglio ad alcune necessità di business. La versione S_1 subirà quindi le seguenti differenze rispetto alla versione S_0:

1. La granularità temporale viene modificata da `Date` a `Month`.

2. Viene inserita nella gerarchia una nuova classificazione in sotto-categorie (`SubCategory`).

3. Un nuovo vincolo di integrità viene modellato nella gerarchia dei clienti, in

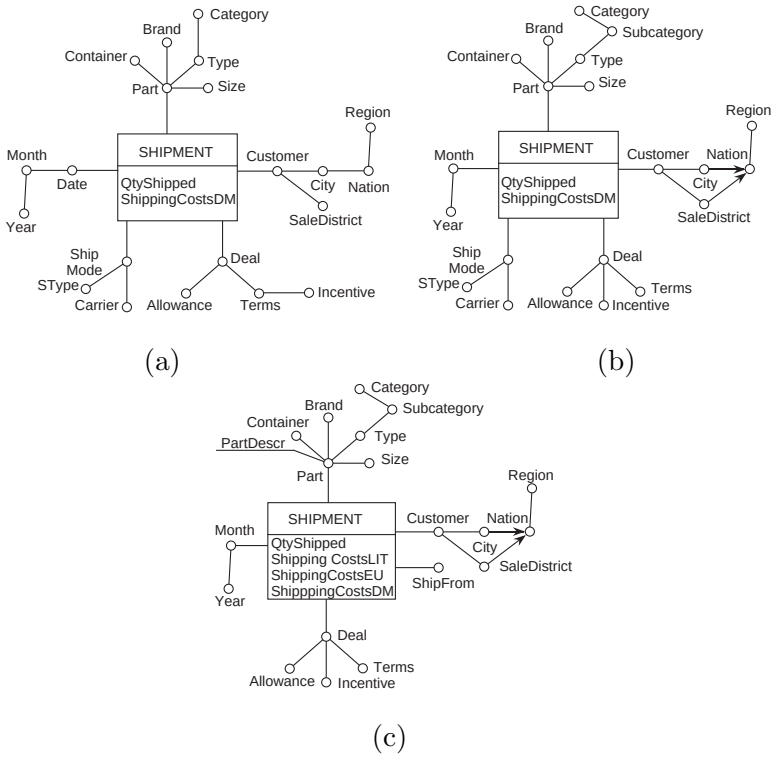


Figura 4.1: Schema Concettuale per le tre versioni del fatto Shipment: S_0 (a), S_1 (b), and S_2 (c)

modo tale che i distretti appartengano ad una Nazione, e per ciascun cliente la nazione del distretto di vendita sia la nazione della città di appartenenza.

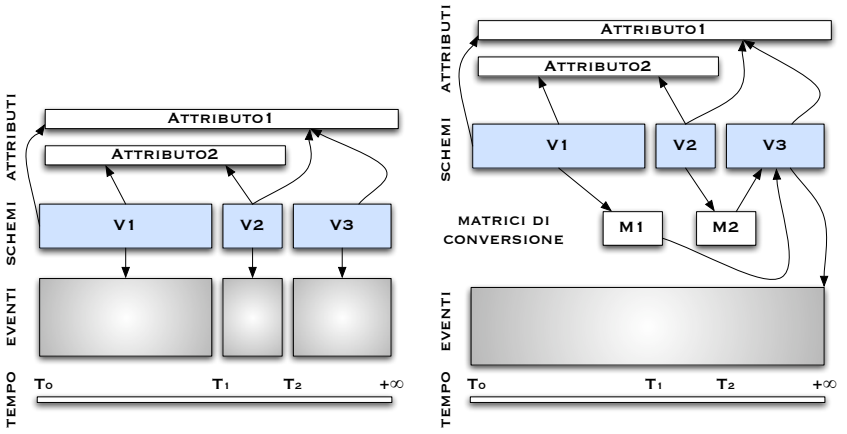
4. L'incentivo (`Incentive`) diventa indipendente dai termini della vendita.

Successivamente, nel momento $t_2 = 1/1/2004$, viene creata un'altra versione S_2:

1. Vengono aggiunte due nuove misure `ShippingCostsEU` e `ShippingCostsLIT`
2. La dimensione `Shipmode` viene eliminata.
3. Viene aggiunta una dimensione `ShipFrom`.
4. Un attributo descrittivo `PartDescr` viene aggiunto a `Part`.

Gli schemi concettuali per le versioni S_1 ed S_2 sono mostrati nella figura 4.1(b,c).

All'interno di un sistema che non supporta le versioni, nel momento di un cambiamento tutti i dati vengono migrati nella nuova versione dello schema. Viceversa



(a) (b)

Figura 4.2: Rapporto tra le versioni e l'insieme degli eventi

se il sistema supporta il versioning, tutti gli schemi precedenti rimarranno ancora a disposizione, per essere eventualmente interrogati insieme ai dati registrati durante il loro periodo di validità. In alcune implementazioni dello schema versioning l'utente ha la possibilità di decidere quale versione dello schema deve essere utilizzato per l'interrogazione dei dati di uno stesso periodo temporale. Questo può accadere solamente se la dimensione temporale delle versioni è completamente indipendente da quella dei dati del cubo multidimensionale, come in [EK01]. Per esempio i dati del 2002 potrebbero essere interrogati attraverso lo schema S_1, introdotto solo nel 2003. In particolare si potrebbe richiedere la distribuzione dei costi sulle sotto-categorie, introdotte nello schema dopo il primo Gennaio del 2003.

Nel nostro caso, invece, verrà utilizzato il partial schema versioning, dato che all'utente non viene permesso l'inserimento dei nuovi dati nelle versioni precedenti all'ultima. Ogni versione avrà un insieme di eventi propri e non sarà possibile interrogare tutto l'insieme degli eventi attraverso una sola versione. Per supplire a questa mancanza verrà introdotta l'operazione di intersezione degli schemi e la possibilità di aumentare le versioni precedenti con le modifiche apportate a quelle nuove. Attraverso questa interfaccia sarà possibile eseguire una query su versioni multiple, su tutti i dati corrispondenti all'intervallo temporale delle versioni coinvolte, eventualmente ricoprendo l'insieme totale degli eventi del data warehouse. Nella figura 4.2 viene mostrata la differenza tra un sistema di versioning nel quale ogni versione è

associata ad un insieme di eventi preciso (a) ed un sistema nel quale ogni versione può essere utilizzata per interrogare tutto l'insieme degli eventi (b). Nel primo caso ad ogni versione Vi corrisponde un determinato intervallo di tempo $[t_i, t_j]$, che coincide con quello degli eventi che sono relativi a Vi. L'insieme degli attributi e delle dipendenze funzionali sarà diverso, ma in generale uno stesso attributo potrà avere una validità temporale che coinvolge più versioni (e le istanze di questo attributo saranno pertanto le stesse). Nel secondo caso tutto l'insieme degli eventi è interrogabile attraverso un insieme di matrici di conversione, che fanno corrispondere i valori di alcuni attributi delle versioni precedenti ad attributi nuovi. E' chiaro che nel secondo caso la gestione delle versioni è virtuale, dal momento che i dati sono immagazzinati secondo una struttura unica, trasformata attraverso le matrici per renderla disponibile anche alle versioni precedenti.

Nell'approccio che abbiamo introdotto, le modifiche che avvengono sullo schema nel corso di operazioni sul data warehouse portano alla creazione di una *history delle versioni dello schema*. Tutte queste versioni sono disponibili per i processi di interrogazione, e la scelta del contesto di formulazione delle query può essere scelto sia in maniera esplicita dall'utente sia in maniera implicita dal sottosistema delle interrogazioni. L'idea chiave è di supportare un sistema flessibile di interrogazioni cross-versionali, che permettono cioè di essere eseguite su versioni diverse dello schema in maniera trasparente per l'utente, a seconda del contesto analitico sul quale desidera lavorare. Per ottenere questo risultato il progettista deve avere la possibilità di arricchire le precedenti versioni utilizzando le informazioni delle modifiche correnti. A questo scopo, quando viene creata una nuova versione il progettista potrà scegliere se creare uno *Schema Aumentato* che estende le versioni precedenti, per riflettere le aggiunte dello schema corrente, sia a livello di schema sia a livello delle istanze. Successivamente definiremo in maniera formale gli *schemi aumentati* e delle operazioni che portano alla loro popolazione, fornendo al contempo un esempio del loro utilizzo.

Per essere più precisi, sia S la versione corrente dello schema ed S' la nuova versione. Dato l'insieme delle differenze tra S ed S', viene proposto in maniera automatica al progettista un insieme delle possibili *Azioni di aumento* da eseguire sui

dati delle versioni precedenti. Queste azioni possono richiedere un controllo sui dati estensionali prima dell'inserimento di nuovi vincoli di integrità oppure l'inserimento di nuove informazioni attraverso l'interazione dell'utente. L'insieme delle azioni che il progettista intende applicare portano alla definizione ed alla popolazione di uno *schema aumentato* S^{AUG}, associato ad S, che verrà utilizzato al posto di S in maniera trasparente per l'utente, per rispondere alle interrogazioni che coinvolgono l'intervallo di validità di S. E' importante notare come S^{AUG} sia sempre una estensione di S, nel senso che l'istanza di S può essere sempre ricavata attraverso una proiezione di S^{AUG}.

Si consideri, ad esempio, le operazioni di modifica dello schema che introducono l'attributo `SubCategory`, eseguito nell'istante $t_1 = 1/1/2003$ per produrre la versione S_1. E' chiaro che per tutti i componenti venduti dopo t_1 (inclusi quelli introdotti dopo t_1 e quelli già esistenti prima di t_1) sia necessario definire una sotto-categoria nel momento della migrazione dei dati, in modo tale che le interrogazioni che coinvolgono `SubCategory` possano essere soddisfatte da t_1 in poi. In ogni caso, se l'utente è interessato alla possibilità di eseguire interrogazioni cross-versionali sugli anni 2002 e 2003, come ad esempio nel caso in cui chieda di interrogare anche i vecchi dati sull'attributo `SubCategory`, è necessario:

1. Definire uno schema aumentato S^{AUG} per S_0 che contiene il nuovo attributo `SubCategory`.
2. Migrare i dati vecchi da S_0 ad S^{AUG}.
3. Assegnare i valori appropriati per `SubCategory` per i vecchi dati in S^{AUG}.

Questo processo permetterà di rispondere alle interrogazioni che coinvolgono `SubCategory` nei vecchi dati attraverso l'istanza di S^{AUG}. Bisogna inoltre notare che mentre i primi due passaggi del procedimento sono interamente gestiti dal sistema del versioning, l'ultimo è di responsabilità del progettista.

Nella figura 4.3 l'aggiunta di un attributo A_1 nello schema V_3 viene propagata negli schemi aumentati corrispondenti alle versioni V_1 e V_2. L'attributo sarà quindi disponibile per tutte le query che coinvolgono l'intervallo temporale $[t_0, \infty]$ o un

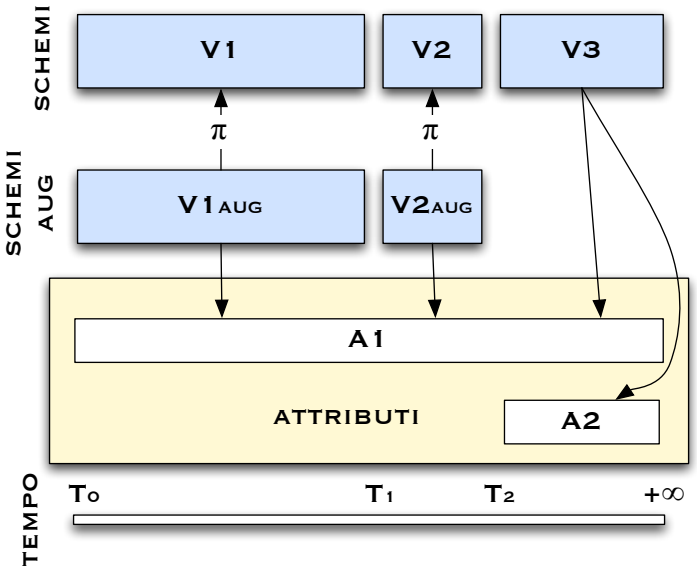


Figura 4.3: Lo Schema Aumentato

suo sotto-intervallo. In alternativa le versioni V_1 e V_2 possono essere interrogate singolarmente ed ottenute attraverso una operazione di proiezione (π) sugli schemi aumentati V_1^{AUG} e V_2^{AUG}.

Come secondo esempio, si consideri l'aggiunta di un nuovo vincolo di integrità tra i distretti di vendita e le nazioni. In questo caso, il progettista può chiedere al sistema di controllare se la dipendenza funzionale tra `SaleDistrict` e `Nation` viene soddisfatta anche sui dati inseriti in passato. Sebbene questa dipendenza non sia stata individuata in fase di progettazione iniziale, potrebbe essere stata soddisfatta sin dal principio, oppure dopo una fase di pulizia dei dati. Quando questo avviene, inserire la nuova dipendenza nello schema aumentato S^{AUG} permette di aumentare le potenzialità delle operazioni di roll-up e drill-down nelle sessioni OLAP.

4.1 Rappresentazione formale degli schemi di data warehousing

Prima di procedere con la rappresentazione formale della soluzione da noi proposta, è necessario richiamare la notazione ed il vocabolario utilizzati. D'ora in avanti utilizzeremo la notazione standard introdotta in [Mai83], dove le lettere maiuscole

a partire dall'inizio dell'alfabeto denotano singoli attributi, mentre quelle a partire dalla fine denotano insiemi. $' \equiv'$ denota l'equivalenza di insiemi di dipendenze funzionali (FD) ed F^+ è la chiusura delle FD in F.

4.1.1 Dipendenze funzionali semplici

Una dipendenza FD $X \rightarrow Y$ è *semplice* se $|X| = |Y| = 1$. Dato un insieme F di dipendenze funzionali semplici su X, diciamo che F è *Aciclico* (rispettivamente *ciclico*) se il grafo orientato (X, F) (cioè il grafo che contiene gli attributi in X come nodi e un arco $(A, B) \forall A \rightarrow B \in F$) è aciclico (rispettivamente ciclico).

Un insieme F di dipendenze funzionali è *canonico* se [Mai83]: [1]

- $\forall$ FD $X \rightarrow Y \in F |Y| = 1$
- F è *ridotto a sinistra*, cioè per ciascuna FD $X \rightarrow A \in F$ non c'è nessun $Y \subsetneqq X$ tale che $Y \rightarrow A \in F$
- F è *non ridondante*, cioè non c'è nessun $F' \subsetneqq F$ tale che $F' \equiv F$.

Per ciascun insieme F di dipendenze funzionali esiste almeno una *Copertura Canonica*, cioè un insieme canonico F^0 di FD tale che $F \equiv F^0$ [Mai83].

4.1.2 Schema graph

Per analizzare il problema del versioning nei data warehouse, dobbiamo introdurre una rappresentazione per gli schemi sulla quale definire le operazioni di modifica. Nel capitolo precedente abbiamo visto alcune rappresentazioni delle dimensioni, chiamate rispettivamente *schema delle dimensioni* [LHV02] (figura 3.6) e *versione di struttura* [EK01] (figura 3.7). Queste due rappresentazioni grafiche hanno il limite di non modellare il fatto e le misure (semplici o derivate), e per definizione rappresentano solamente dimensioni i cui grafi sono aciclici. Introduciamo quindi una nuova rappresentazione basata sui grafi, chiamata *Schema Graph*, che cattura le informazioni chiave dei modelli multidimensionali (come ad esempio il DFM). In maniera

[1] Gli insiemi canonici sono chiamati *minimi* in [Ull88], mentre la nozione di minimalità in [Mai83] ha un significato differente.

intuitiva, in coerenza con [LV03], uno schema data warehouse è un grafo orientato, dove i nodi sono attributi (sia proprietà che misure), e gli archi rappresentano le dipendenze funzionali semplici di una copertura canonica. La rappresentazione degli schemi dei data warehouse in termini di grafi ci permette di definire le modifiche agli schemi attraverso le quattro operazioni di modifica dei grafi:

1. Aggiunta di un nodo (attributo)
2. Rimozione di un nodo (attributo)
3. Aggiunta di un arco (dipendenza funzionale)
4. Rimozione di un arco (dipendenza funzionale)

Questo modello ci permette di semplificare l'analisi degli schemi, grazie ad una visualizzazione intuitiva e confortevole.

Definiamo quindi in maniera formale gli *Schema Graph*:

Definizione 13 (Schema Graph) *Uno* schema graph *è un grafo orientato* $S = (\hat{U}, F)$ *con nodi* $\hat{U} = \{E\} \cup U$ *ed archi* F*, dove*

1. E *è chiamato* fact node *e rappresenta il fatto di interesse. Le sue istanze sono i singoli eventi che sono stati registrati, cioè le singole tuple della tabella del fatto (fact table);*
2. U *è un insieme di attributi (incluse le proprietà e le misure);*
3. F *è un insieme di FD semplici definite su* $\{E\} \cup U$ *nello schema del data warehouse;*
4. E *ha solo archi in uscita, ed esiste un percorso che parte da* E *e termina in ciascun attributo in* U*.*

S *è chiamato schema graph* canonico *se* F *è canonico.*

Gli schema graph canonici per il fatto `shipment` della figura 4.1 sono mostrati nelle figure 4.4 – 4.5 – 4.6. Come possiamo vedere anche dagli esempi, ogni schema graph rappresenta una versione, a differenza delle *versioni di struttura* introdotti in [EK01]. Archi e nodi, quindi, non vengono etichettati con gli intervalli di validità,

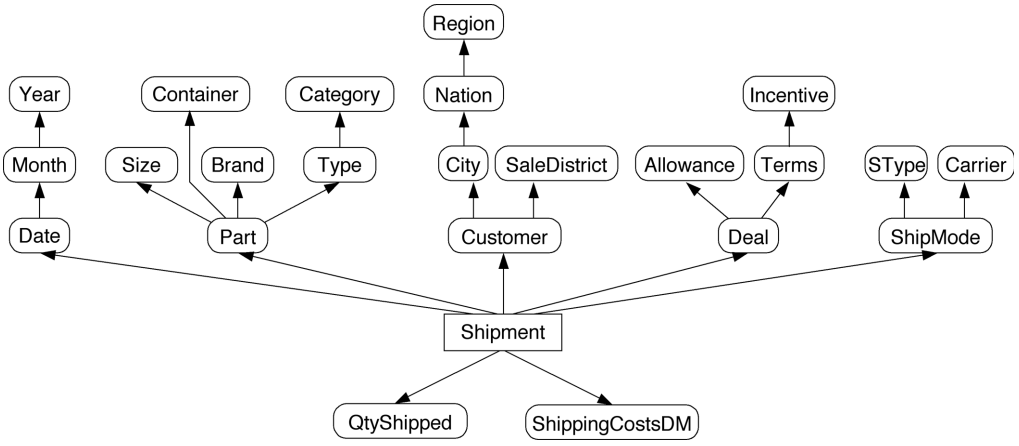


Figura 4.4: Schema graph S_0

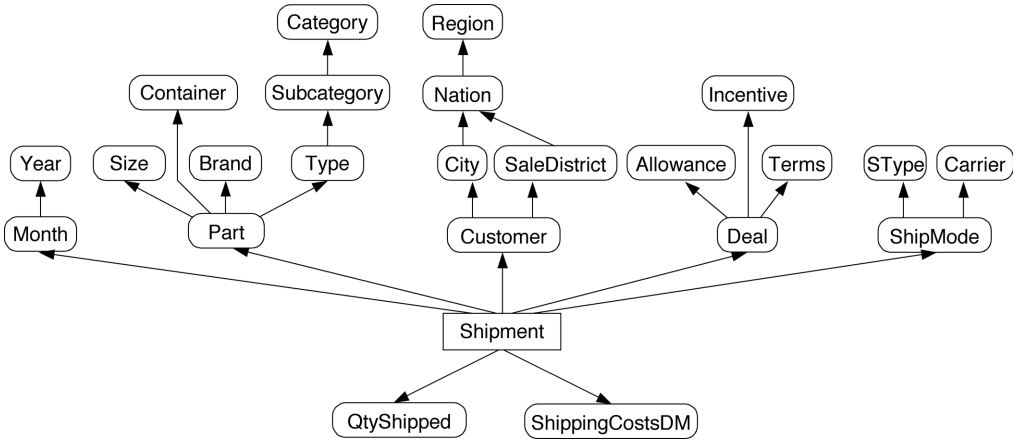


Figura 4.5: Schema graph S_1

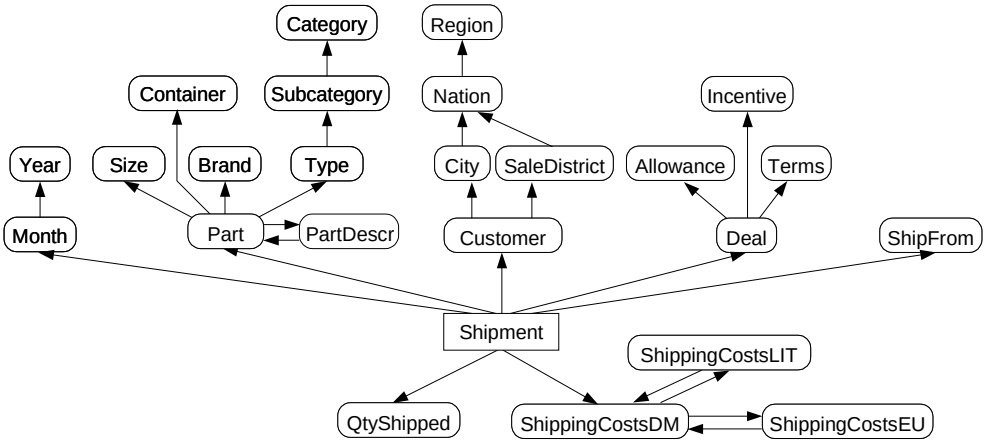


Figura 4.6: Schema graph S_2

perché questi sono già determinati dalla definizione della versione corrispondente al grafo, che vedremo successivamente.

Gli schema graph così definiti soddisfano l'*Universal Relation Schema Assumption* (URSA) [Mai83], che è una assunzione standard nella progettazione dei database relazionali. L'URSA connette il nome di un attributo alla sua semantica, cioè stabilisce che all'interno dell'insieme degli schema graph che descrivono lo schema del data warehouse tutte le occorrenze del nome di un attributo hanno obbligatoriamente lo stesso significato. Nel nostro esempio, quindi, i due differenti concetti rappresentati dal tipo di componenti e dal tipo di vendita sono dichiarati attraverso due nomi distinti (`Type` ed `SType`). D'altra parte è possibile che un attributo appaia con lo stesso nome in versioni diverse dello stesso schema o in schemi diversi, se modella lo stesso concetto.

Infine dobbiamo far notare che, in riferimento al modello multidimensionale, una dipendenza funzionale $f \in F$ ha un impatto sulla semantica degli attributi, nella maniera seguente:

1. $f = E \rightarrow A$

   - A può essere una dimensione. Dal momento che i valori di E rappresentano gli eventi singoli, in questo caso f esprime il fatto che ciascun evento è connesso ad esattamente un solo valore per ciascuna dimensione.
   - A può essere una misura. In questo caso f rappresenta il fatto che ciascun evento è associato ad esattamente un valore per ciascuna misura.

2. $f = B \rightarrow A$

   - B può essere una dimensione o una proprietà, ed A è una proprietà. In questo caso, $B \rightarrow A$ modella una associazione molti-a-uno all'interno di una gerarchia (per esempio B può essere una città ed A una nazione). Dal punto di vista dell'implementazione questo significa che la corrispondenza tra ogni valore di B ed esattamente un valore di A è registrata in maniera esplicita nel database (per esempio, all'interno della tabella della dimensione di uno star schema).
   - B può essere una misura, ed A una misura derivata. In questo caso, $B \rightarrow A$ modella il fatto che A possa essere derivata da B. Dal punto di

vista implementativo, esistono due scelte possibili: si può immagazzinare A come un attributo separato, che introduce delle informazioni ridondanti nel data warehouse ma può avere vantaggi di velocità se la complessità computazionale del calcolo di A da B è complessa, oppure si può immagazzinare una funzione nel repository dei meta-dati permettendo il calcolo al volo di A a partire da B.

Detto questo, risulta chiaro che dalla rappresentazione grafica appena introdotta non sia possibile stabilire se un attributo sia (1) una dimensione o una misura, oppure (2) una proprietà o una misura derivata. Sarà quindi necessario salvare queste informazioni nel deposito dei meta-dati, come vedremo in seguito.

4.1.3 Schema graph ridotto

Confrontato con gli schema graph generali, la classe degli schema graph canonici ha l'importante vantaggio di fornire una rappresentazione non ridondante e compatta. D'altra parte, allo scopo di ottenere dei risultati univoci per le operazioni di modifica degli schemi, dobbiamo assicurarci di lavorare su rappresentazioni *univocamente determinate*. Questa sezione dimostra come sia possibile ottenere uno schema determinato univocamente, chiamato *Schema Graph Ridotto*. Per gli insiemi aciclici di FD semplici, le coperture canoniche sono determinate univocamente e possono essere calcolate attraverso una riduzione transitiva:

Definizione 14 (Chiusura Transitiva) *Sia U un insieme di attributi, ed F un insieme di FD semplici su U. La* chiusura transitiva *di F, denominata F^*, è definita in maniera induttiva come segue:*

1. $f \in F \Longrightarrow f \in F^*$
2. $A \rightarrow B \in F^* \ \wedge \ B \rightarrow C \in F^* \Longrightarrow A \rightarrow C \in F^*$

Definizione 15 (Riduzione Transitiva) *Sia U un insieme di attributi, ed F un insieme di FD semplici su U. La* riduzione transitiva *di F è l'insieme minimo F^- di FD semplici su U tale che $F^* = (F^-)^*$.*

Per illustrare le differenze tra la chiusura transitiva F^* e la chiusura F^+ delle dipendenze funzionali F, osserviamo che la definizione di F^* contiene solo FD

semplici. Se le FD in F coinvolgono due o più attributi allora F^+ contiene delle dipendenze funzionali non semplici addizionali che sono implicite in F^*.

Esempio 3 *Per* $F = \{A \rightarrow B, B \rightarrow C\}$ *abbiamo:*

$$\begin{aligned} F^* &= \{A \rightarrow B, B \rightarrow C, A \rightarrow C\} \\ F^+ &= \{A \rightarrow B, B \rightarrow C, A \rightarrow C, \\ &\quad A \rightarrow A, AB \rightarrow A, AC \rightarrow A, ABC \rightarrow A, \\ &\quad B \rightarrow B, AB \rightarrow B, AC \rightarrow B, BC \rightarrow B, ABC \rightarrow B, \\ &\quad C \rightarrow C, AB \rightarrow C, AC \rightarrow C, BC \rightarrow C, ABC \rightarrow C\} \end{aligned}$$

Come visto in [AGU72], nel caso dei grafi aciclici la riduzione transitiva F^- è univocamente determinata ed $F^- \subseteq F$. Oltre a questo, ricordiamo che F^- è una copertura canonica di F, da [Lec04], quindi se F è aciclico lo schema graph ridotto per $S = (\hat{U}, F)$ è $S^- = (\hat{U}, F^-)$.

D'altra parte, negli scenari reali l'aciclicità delle dipendenze funzionali può non essere soddisfatta, dal momento che i cicli si presentano in almeno due casi:

- *Proprietà Descrittive.* Sebbene nella quasi totalità dei casi le associazioni tra le proprietà di un data warehouse abbiano una molteplicità molti-a-uno, in alcuni casi possono avere una molteplicità uno-ad-uno. Questo accade tipicamente con le proprietà descrittive, che permettono di descrivere un oggetto attraverso diverse nomenclature (per esempio i codici dei prodotti ed il loro nome).

- *Misure Derivate.* Due misure derivate possono essere calcolabili l'una a partire dall'altra, se la funzione di derivazione è invertibile. Questo accade, per esempio, nel caso della conversione dei prezzi da una valuta all'altra, semplicemente applicando dei fattori costanti di conversione.

L'esempio seguente dimostra che le coperture canoniche non sono più univoche quando l'insieme delle dipendenze funzionali è ciclico.

Esempio 4 *Consideriamo le misure* A_1 *ed* A_2 *i cui valori sono derivabili l'una dell'altra. Avremo quindi un ciclo di dipendenze funzionali composto da* $A_1 \rightarrow A_2$ *e* $A_2 \rightarrow A_1$*. Consideriamo ora una misura* A_3 *che può essere derivata da* A_1 *e da* A_2*. A questo punto, abbiamo un insieme* F *di dipendenze funzionali*

$\{A_1 \rightarrow A_2, A_2 \rightarrow A_1, A_1 \rightarrow A_3, A_2 \rightarrow A_3\}$, *ed è facilmente dimostrabile che* $\{A_1 \rightarrow A_2, A_2 \rightarrow A_1, A_1 \rightarrow A_3\}$ *e* $\{A_1 \rightarrow A_2, A_2 \rightarrow A_1, A_2 \rightarrow A_3\}$ *sono entrambe coperture canoniche di* F.

E' necessario quindi dimostrare la possibilità di determinare in maniera univoca la forma ridotta per gli schema graph anche in presenza di dipendenze funzionali cicliche [Lec04]. Consideriamo uno schema graph $S = (\hat{U}, F)$, dove F non è né ciclica né canonica. La relazione $\equiv_F$ su $\hat{U}$, definita come:

$$A \equiv_F B \quad \text{se} \quad A \rightarrow B \in F^+ \wedge B \rightarrow A \in F^+$$

per $A, B \in \hat{U}$ è una relazione di equivalenza. $\hat{U}/_{\equiv_F}$ viene utilizzata per denotare l'insieme delle classi di equivalenza indotte da $\equiv_F$ su $\hat{U}$. Consideriamo quindi il grafo aciclico orientato dove ciascun nodo è una classe di equivalenza $X \in \hat{U}/_{\equiv_F}$ ed un arco va da X ad Y, con $X \neq Y$, se esistono degli attributi $A \in X, B \in Y$ tali che $A \rightarrow B \in F$. La riduzione transitiva di questo grafo è lo *schema graph aciclico equivalente* per S e denominato da $S^a = (\hat{U}/_{\equiv_F}, F^a)$. S^a è aciclico (data la sua costruzione) ed univocamente determinato (dato che la riduzione transitiva è univoca per i grafi aciclici). Sia ora $<_S$ un ordinamento totale su $\hat{U}$ (specificato dall'utente o generato dal sistema su alcuni criteri di ordinamento come il timestamp di creazione dell'attributo o il nome dell'attributo).

Definizione 16 (Dipendenze Funzionali Implicite) *Sia* $\hat{U}$ *un insieme di attributi per i quali esiste un ordinamento totale* $<_S$, F *un insieme (eventualmente ridondante e/o ciclico) di dipendenze funzionali semplici su* $\hat{U}$, *ed* $X = \{A_1, \ldots, A_n\} \in \hat{U}/_{\equiv_F}$, $n \geq 1$. *Sia* $A_1 <_S A_2 <_S \ldots <_S A_n$ *l'ordinamento degli attributi in* X *ottenuti con* $<_S$. *Le* dipendenze funzionali implicite (dato $<_S$) *sono date da* $F_X = \{A_1 \rightarrow A_2, A_2 \rightarrow A_3, \ldots, A_{n-1} \rightarrow A_n, A_n \rightarrow A_1\}$.

Definizione 17 (Schema Graph Ridotto) *Sia* $S = (\hat{U}, F)$ *uno schema graph ed* $S^a = (\hat{U}/_{\equiv_F}, F^a)$ *sia lo schema graph aciclico equivalente per* S. *Lo* schema graph ridotto per S *è il grafo orientato* $S^- = (\hat{U}, F^-)$, *dove*

$$F^- = \bigcup_{X \rightarrow Y \in F^a} \{\min X \rightarrow \min Y\} \cup \bigcup_{X \in \hat{U}/_{\equiv_F}} F_X.$$

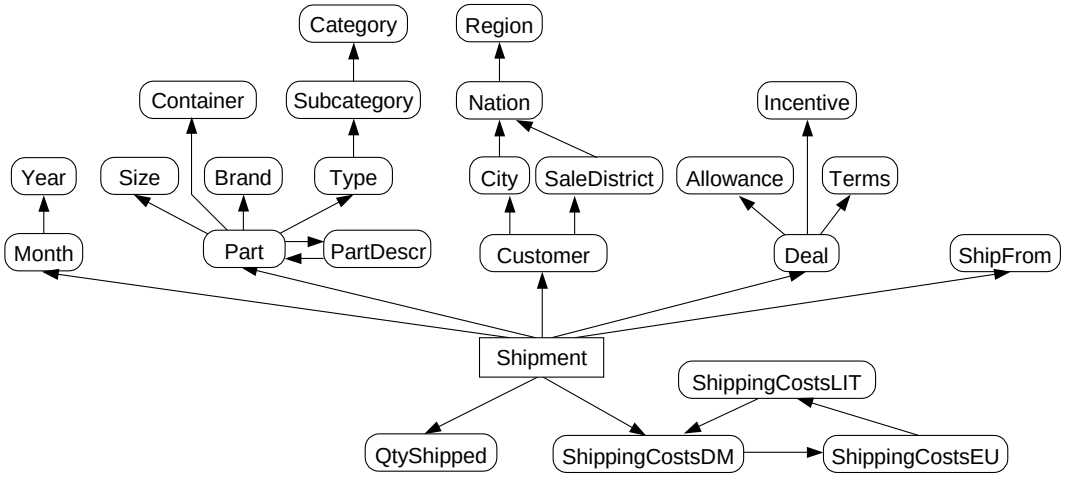


Figura 4.7: Forma ridotta per lo schema schema graph S_2

Da [AGU72, Lec04] sappiamo che lo schema graph ridotto $S^- = (\hat{U}, F^-)$ per S è una riduzione transitiva univocamente determinata per S, ed F^- è una copertura canonica per F. D'ora in avanti, dato un insieme F di dipendenze funzionali semplici, useremo $'-'$ per denotare l'operatore di riduzione che produce la copertura canonica univocamente determinata F^- di F, secondo la definizione 17.

Esempio 5 *Consideriamo lo schema graph S_2 nella figura 4.6, dove* `Part` *ha una proprietà equivalente* `PartDescr` *ed i costi di spedizione sono espressi in Euro, Lire e Marchi. La forma ridotta per S_2, basata sull'ordinamento totale indotto dai nomi degli attributi, è mostrato nella figura 4.7.*

4.1.4 Proiezione sugli schema graph

Dato un insieme F di dipendenze funzionali sugli attributi U e dato $X \subseteq U$, la *proiezione di F su X* è data da $\pi_X(F) := \{A_1 \to A_2 \in F^+ \mid A_1A_2 \subseteq X\}$. Basandoci sui risultati in [LV03] e [Lec04], in questa sezione dimostreremo che l'operazione di proiezione π è chiusa sugli schema graph. Questo risultato è importante perché ci permetterà di definire con precisione l'effetto della cancellazione di un attributo da uno schema graph.

Il nostro scopo è quello di mantenere il maggior numero di informazioni possibile sulle dipendenze funzionali durante le operazioni di cancellazione degli attributi.

Dato che le proiezioni sono definite come un sottoinsieme della chiusura, esistono diverse problematiche sull'applicazione diretta di π:

1. In generale la proiezione di un insieme di dipendenze funzionali coinvolge dipendenze funzionali non semplici, che sono al di fuori del nostro studio

2. Il risultato della chiusura può crescere esponenzialmente con l'aumentare degli attributi coinvolti.

Lemma 1 ([LV03]) *Sia U un insieme di attributi, F un insieme di dipendenze funzionali semplici su U, e sia $A \in U$ ed $X \subseteq U$ tali che $X \neq \emptyset$, $A \notin X$, e $X \to A \in F^+$. Allora esiste una sequenza di $n \geq 2$ attributi $A_1, \ldots, A_n \in U$ tale che $A_1 \in X$, $A_n = A$, e $A_i \to A_{i+1} \in F$, $1 \leq i \leq n-1$.*

Lemma 2 ([Lec04]) *Sia U un insieme di attributi, F un insieme di dipendenze semplici su U, ed $X \subseteq U$. Sia $F_X = \pi_X(F)^0$ e $F'_X = \{A_1 \to A_2 \in F^* \mid A_1A_2 \subseteq X\}^-$.*

1. *F_X contiene solo dipendenze funzionali semplici*

2. *$F_X \equiv F'_X$.*

Visto il lemma 2, d'ora in avanti daremo per assunto che per un insieme F di dipendenze funzionali semplici, la proiezione è definita come $\pi_X(F) := \{A_1 \to A_2 \in F^* \mid A_1A_2 \subseteq X\}$, che è per definizione un insieme di dipendenze funzionali semplici.

Teorema 1 *Sia $S = (\hat{U}, F)$ uno schema graph, ed $X \subseteq \hat{U}$ tale che $E \in X$. Allora $(X, \{A_1 \to A_2 \in F^* \mid A_1A_2 \subseteq X\}^-)$ è uno schema graph ridotto.*

4.2 Algebra delle modifiche degli schemi

Come abbiamo visto nelle soluzioni presentate fino ad oggi per la gestione del versioning nei data warehouse, è necessario prima di tutto stabilire un'algebra delle operazioni di modifica, affinché si possano calcolare sulla base di questo insieme di operazioni di base tutte le operazioni possibili. L'introduzione di una rappresentazione dei cubi multidimensionali basata su grafi orientati ci permette di semplificare considerevolmente l'insieme delle operazioni di base: $\texttt{Add}_\texttt{A}, \texttt{Del}_\texttt{A}, \texttt{Add}_\texttt{F}, \texttt{Del}_\texttt{F}$, rispettivamente per aggiungere un attributo ed eliminarlo, aggiungere una dipendenza

funzionale semplice ed eliminarla. Per ognuna di queste operazioni definiremo gli effetti sullo schema graph, che d'ora in avanti considereremo sempre nella sua forma ridotta, utilizzando i termini schema graph ed schema graph ridotto indistintamente.

In aggiunta alle quattro operazioni appena introdotte, assumiamo che ci sia una operazione per creare uno schema iniziale $S = (\{E\}, \emptyset)$, che contiene solo il fact node, ed una che permette di cancellare uno schema esistente.

Sia $S = (\hat{U}, F)$ uno schema graph. Per ciascuna operazione di modifica $M(Z)$ (dove M è $\texttt{Add}_\texttt{A}$ o $\texttt{Del}_\texttt{A}$ e Z è un attributo, oppure M è $\texttt{Add}_\texttt{F}$ o $\texttt{Del}_\texttt{F}$ e Z è una dipendenza funzionale. Definiamo il nuovo schema $\texttt{New}(S, M(Z))$ ottenuto applicando $M(Z)$ sullo schema corrente S.

Definizione 18 *Sia $S = (\hat{U}, F)$ uno schema graph ridotto, ed A un attributo. Allora:*

$$\texttt{New}(S, \texttt{Add}_\texttt{A}(A)) := (\hat{U} \cup \{A\}, (F \cup \{E \rightarrow A\})^-)$$

Nella definizione 18 non distinguiamo i casi in cui A sia già presente o meno in $S = (\hat{U}, F)$. Infatti se A è già presente in S lo schema risultante rimarrà invariato, mentre se non lo è verrà direttamente connesso attraverso un arco al fact node E. [2]

Al momento dell'inserimento il progettista dovrà specificare se questo attributo è una proprietà o una misura. Queste informazioni verranno registrate nei meta-dati.

Definizione 19 *Sia $S = (\hat{U}, F)$ uno schema graph ridotto, e sia $A \in U$ un attributo. Allora:*

$$\texttt{New}(S, \texttt{Del}_\texttt{A}(A)) := (\hat{U} \setminus \{A\}, \pi_{\hat{U} \setminus \{A\}}(F)^-)$$

Quindi, secondo il teorema 1 la cancellazione di una attributo A è definita rimuovendo A e mantenendo le dipendenze funzionali che non coinvolgono A attraverso la proiezione π

Definizione 20 *Sia $S = (\hat{U}, F)$ uno schema graph ridotto, ed $f = A_1 \rightarrow A_2$ una dipendenza funzionale che coinvolge almeno un attributo in U. Allora avremo:*

$$\texttt{New}(S, \texttt{Add}_\texttt{F}(f)) := (\hat{U}, (F \cup \{f\})^-)$$

[2] Ricordiamo che il sistema soddisfa l'assunzione che i nomi degli attributi siano univoci (URSA) [Mai83], quindi non potranno coesistere due attributi con lo stesso nome all'interno di uno stesso schema, oppure all'interno di due schemi differenti con lo stesso significato.

Bisogna notare che l'introduzione di una nuova dipendenza funzionale può introdurre ridondanze, che vengono rimosse attraverso la riduzione "−".

Definizione 21 *Sia $S = (\hat{U}, F)$ uno schema graph ridotto, ed $f = A_1 \rightarrow A_2$ una dipendenza funzionale già esistente in F, dove $A_1 \neq E$. Sia*

$$\begin{aligned} F' &= F \setminus \{f\} \\ &\quad \cup \{A_0 \rightarrow A_2 \mid (\exists A_0 \in \hat{U})\ A_0 \rightarrow A_1 \in F\} \\ &\quad \cup \{A_1 \rightarrow A_3 \mid (\exists A_3 \in U)\ A_2 \rightarrow A_3 \in F\} \end{aligned}$$

Allora:

$$\texttt{New}(S, \texttt{Del}_\texttt{F}(f)) := (\hat{U}, (F')^-)$$

La definizione 21 afferma che la dipendenza funzionale $f = A_1 \rightarrow A_2$ viene eliminata attraverso la differenza degli insiemi. Successivamente vengono inserite in F tutte le dipendenze funzionali che determinano A_1 e tutte le dipendenze funzionali che partono da A_1. Questa definizione permette di gestire qualsiasi operazione di cancellazione di dipendenze funzionali semplici, senza nessuna perdita di informazioni importanti implicitamente connesse a quella che stiamo eliminando dallo schema.

Esempio 6 *La sequenza delle operazioni applicate ad S_0 per arrivare ad S_1 è* $\texttt{Del}_\texttt{A}(\textit{Date})$, $\texttt{Add}_\texttt{A}(\textit{SubCategory})$, $\texttt{Add}_\texttt{F}(\textit{Type} \rightarrow \textit{SubCategory})$, $\texttt{Add}_\texttt{F}(\textit{SubCategoryCategory})$, $\texttt{Add}_\texttt{F}(\textit{SaleDistrict} \rightarrow \textit{Nation})$, $\texttt{Del}_\texttt{F}(\textit{Terms} \rightarrow \textit{Incentive})$ *In particolare, gli schema graph ridotto risultanti per la gerarchia della dimensione* **Part** *dopo ciascuna operazione nella seconda linea sono mostrate nella figura 4.8: (a) mostra lo schema graph dopo l'inserimento del nuovo attributo* SubCategory, *(b) dopo l'inserimento del nuovo arco* $\textit{Type} \rightarrow \textit{SubCategory}$, *e (c) dopo l'inserimento del nuovo arco* $\textit{SubCategory} \rightarrow \textit{Category}$

La sequenza delle operazioni applicate ad S_1 per arrivare ad S_2 è: $\texttt{Add}_\texttt{A}(\textit{PartDescr})$, $\texttt{Add}_\texttt{F}(\textit{Part} \rightarrow \textit{PartDescr})$, $\texttt{Add}_\texttt{F}(\textit{PartDescr} \rightarrow \textit{Part})$, $\texttt{Add}_\texttt{A}(\textit{ShipFrom})$, $\texttt{Del}_\texttt{A}(\textit{SType})$, $\texttt{Del}_\texttt{A}(\textit{Carrier})$, $\texttt{Del}_\texttt{A}(\textit{ShipMode})$, $\texttt{Add}_\texttt{A}(\textit{ShippingCostsEU})$, $\texttt{Add}_\texttt{A}(\textit{ShippingCostsLIT})$, $\texttt{Add}_\texttt{F}(\textit{ShippingCostsDM} \rightarrow \textit{ShippingCostsEU})$,

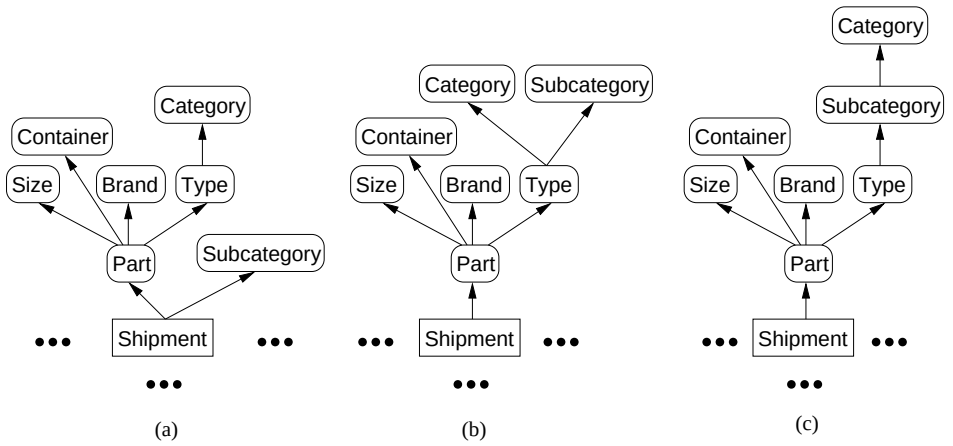


Figura 4.8: Schema Graph ridotto per la gerarchia della dimensione `Part` dopo le operazioni di modifica.

$\texttt{Add}_\texttt{F}(\textit{ShippingCostsEU} \rightarrow \textit{ShippingCostsDM})$,
$\texttt{Add}_\texttt{F}(\textit{ShippingCostsDM} \rightarrow \textit{ShippingCostsLIT})$,
$\texttt{Add}_\texttt{F}(\textit{ShippingCostsLIT} \rightarrow \textit{ShippingCostsDM})$

Possiamo quindi dimostrare che le definizioni 18–21 formalizzano il risultato atteso per le modifiche degli schemi, ed in particolare che queste operazioni di modifica sono chiuse sugli schema graph.

Teorema 2 *Sia $S = (\hat{U}, F)$ uno schema graph ridotto.*

1. *Sia $(\hat{U}', F') = \texttt{New}(S, \texttt{Add}_\texttt{A}(A))$. Allora $A \in \hat{U}'$.*
2. *Sia $(\hat{U}', F') = \texttt{New}(S, \texttt{Del}_\texttt{A}(A))$. Allora $A \notin \hat{U}'$.*
3. *Sia $(\hat{U}', F') = \texttt{New}(S, \texttt{Add}_\texttt{F}(f))$. Allora $f \in F'^{+}$.*
4. *Sia $(\hat{U}', F') = \texttt{New}(S, \texttt{Del}_\texttt{F}(f))$. Allora $f \notin F'^{+}$.*

In aggiunta, in tutti questi casi $(\hat{U}', F')$ è uno schema graph ridotto.

> **Dimostrazione:** La (1), (2), e la (3) sono dirette conseguenze delle definizioni 18, 19, e 20, rispettivamente. La (4) è conseguente dalla definizione 21, osservando che F è canonica e quindi in particolare è non ridondante. Rimane da dimostrare che $(\hat{U}', F')$ è uno schema graph ridotto, cioè che: (a) F' è un insieme di dipendenze funzionali semplici

(b) E ha solo archi in uscita ed esiste un percorso da E a ciascun attributo di $\hat{U}'$ (c) $(\hat{U}', F')$ è in forma ridotta. Per $\texttt{Add}_\texttt{A}$, $\texttt{Add}_\texttt{F}$, e $\texttt{Del}_\texttt{F}$, (a) e (b) sono banali. Per $\texttt{Del}_\texttt{A}$, (a) e (b) sono conseguenti dal teorema 1. Per tutte le quattro operazioni, (c) dipende direttamente dalla definizione dell'operatore di riduzione "$-$".

Questo teorema è necessario per dimostrare che le operazioni di modifica che abbiamo appena definito portano a schemi validi e viene garantita la produzione di risultati non ridondanti ed univocamente determinati.

4.3 Versioni

Come abbiamo già visto nel capitolo 3, dedicato al problema del tempo nei data warehouse, nello schema versioning l'utente o il sistema stabiliscono la *validità temporale degli schemi*, secondo criteri di consistenza e di corrispondenza con la realtà che è necessario modellare. Uno schema che riflette correttamente le necessità di business durante un periodo temporale definito, viene etichettato attraverso un identificativo di versione ed un *Periodo di Validità Temporale*. Nell'approccio che andiamo a presentare non c'è possibilità di sovrapposizione temporale tra due schemi, e risulta quindi impossibile modellare versioni alternative definite in uno stesso istante t_1, come invece avviene in [BEK+04].

La validità temporale di una versione creata all'istante t_0 è l'intervallo $[t_0, +\infty]$ se è l'ultima versione che è stata definita, mentre è $[t_0, t_1]$, dove t_1 è l'istante di creazione della versione successiva.

Ogni versione è popolata dagli eventi che sono accaduti in questo periodo di tempo, e su questo possono essere interrogati.

In accordo con [MS90] nel caso del versioning degli schemi non c'è nessuna necessità di distinzione tra valid-time e transaction-time.

Una nuova versione è il risultato di una serie di operazioni di modifica, che chiamiamo *Transazione di Modifiche sullo Schema*, o semplicemente *transazione*. In analogia ai concetti usuali di transazione, i risultati intermedi ottenuti attraverso le singole operazioni di modifica degli schemi sono invisibili ai fini dell'interrogazione. Oltre a questo gli schemi di modifica intermedi non sono popolati di eventi, e ad essi non viene associato nessuno schema aumentato.

Una transazione produce una nuova versione ed uno schema aumentato per ciascuna versione precedente, ognuna delle quali viene (fisicamente o virtualmente) popolata di dati.

4.3.1 Migrazione dei dati

Data una versione S, sia $M_1(Z_1), \ldots, M_h(Z_h)$ la sequenza di operazioni eseguite all'interno di una transazione di modifiche sullo schema. La nuova versione S' è definita eseguendo le operazioni di modifica una dopo l'altra.

Allo scopo di popolare S', è necessario portare a termine alcune operazioni di migrazione che sposteranno i dati da una versione alla successiva. Se S' viene creata all'istante t, le operazioni di migrazione coinvolgono i dati che sono validi al tempo t, quindi tutti i dati la cui validità coinvolge sia S sia S'. Nei data warehouse le considerazioni sulla validità dei dati possono essere distinte in due categorie:

- **Eventi**: Gli eventi avvengono in un particolare istante di tempo. In accordo con [J+98], assumiamo che la validità di un evento che avviene all'istante t_e è il periodo di tempo di lunghezza zero $[t_e, t_e]$. In questo modo, un evento non può essere presente in due diverse versioni. Quando una nuova versione viene creata, non è quindi necessario eseguire nessuna operazione di migrazione degli eventi presenti nella fact table. Questa considerazione mostra una grossa differenza con la soluzione dello *schema evolution*, nel quale tutti gli eventi vengono migrati in accordo con il nuovo schema (che sostituisce in tutto e per tutto i precedenti).

- **Istanze di gerarchie**: Le istanze delle gerarchie di proprietà sono generalmente valide per intervalli di tempo $[t_1, t_2]$, quindi la loro validità può sovrapporsi a diverse versioni dello schema. Al momento della creazione di una nuova versione S', quindi, possono essere necessarie diverse operazioni di migrazione delle istanze ancora valide per S'.

Per determinare l'insieme delle operazioni di migrazione che devono essere eseguite dopo una transazione, indipendentemente dalla sequenza delle operazioni coinvolte all'interno di questa, definiamo l'*effetto di rete* per gli attributi e le dipendenze funzionali connesse alla transazione. Sia $S' = (\hat{U}', F')$ la nuova versione ottenuta

Tabella 4.1: Azioni di migrazione associate alle operazioni di aggiunta o rimozione di attributi e dipendenze funzionali

Elemento	*Condizione*		*Azione di Migrazione*
$A \in \texttt{Diff}_\texttt{A}^+(S, S')$	$(E \to A) \notin F'$	A è una proprietà	Aggiungere i valori per A
$A \in \texttt{Diff}_\texttt{A}^-(S, S')$	$(E \to A) \notin F'$	A è una proprietà	Eliminare A
$f \in \texttt{Diff}_\texttt{F}^+(S, S')$	-		Controllare f

applicando la transazione alla versione $S = (\hat{U}, F)$. Definiamo quindi:

$\texttt{Diff}_\texttt{A}^+(S, S') := U' \setminus U$ (insieme degli attributi aggiunti)

$\texttt{Diff}_\texttt{F}^+(S, S') := F' \setminus F^*$ (insieme delle dipendenze funzionali aggiunte)

$\texttt{Diff}_\texttt{A}^-(S, S') := U \setminus U'$ (insieme degli attributi eliminati

Non viene definito $\texttt{Diff}_\texttt{F}^-(S, S')$ (insieme delle dipendenze funzionali eliminate), dal momento che a seguito di una cancellazione di una dipendenza funzionale non è necessario eseguire nessuna operazione. Quando viene eliminata una dipendenza funzionale una regola di integrità viene eliminata, ma non è necessario nessun allineamento delle istanze precedenti.

Aggiungere i valori per A: Una nuova proprietà A è stata aggiunta alla gerarchia. Ogni istanza valida della gerarchia viene migrata in S', inserendo i valori per A. Come esempio, al momento dell'inserimento di `SubCategory` dovranno essere inseriti tutti i valori validi per la gerarchia della dimensione `Part`.

Eliminare A: Se la proprietà A viene cancellata, i suoi valori sono semplicemente eliminati da tutte le istanze valide migrate in S'

Controllare f: Una nuova dipendenza funzionale f è stata aggiunta alla gerarchia. Avremo quindi due casi distinti:

1. Se la nuova dipendenza funzionale f coinvolge due attributi già presenti, è necessario controllare se f è soddisfatta per tutte le istanze valide esistenti, ed eventualmente forzare questo vincolo modificando sotto la guida dell'utente i valori di uno o entrambi gli attributi coinvolti da f.

2. Se f coinvolge una nuova proprietà A appena aggiunta, allora sarà necessario solamente assicurarsi che tutte le nuove istanze soddisfino f.

In tutti i casi non citati nella tabella 4.1 non è necessario eseguire nessuna operazione di migrazione. Per quanto riguarda la cancellazione o l'aggiunta di una misura, infatti, vengono coinvolti solamente le istanze del fatto, e dato che gli eventi non possono essere presenti contemporaneamente in due versioni distinte, non ci sarà nessuna migrazione. La cancellazione di una dipendenza funzionale non ha nessun impatto sulle istanze delle gerarchie, in quanto un vincolo precedentemente presente viene a mancare, ma i valori delle istanze rimangono tali e quali.

4.3.2 Aggiornamento degli schemi

Per permettere una maggiore flessibilità e potenza nella realizzazione di query che interrogano basi di dati corrispondenti a versioni diverse dello schema vengono introdotti gli *Schemi Aumentati*, che rappresentano un aggiornamento degli schemi precedenti. Gli *Schemi Aumentati* sono schemi che vengono prodotti attraverso insiemi di operazioni di modifica, riportate nelle versioni precedenti per allargare i limiti del range delle interrogazioni multi-schema. Ad ogni versione dello Schema Graph corrisponde un diverso schema aumentato, se il progettista decide di riportare in quel determinato schema una o più modifiche apportate all'ultima versione.

Una transazione di operazioni di modifica porta ad ottenere una versione $S' = (\hat{U}', F')$ a partire da $S = (\hat{U}, F)$. Sia S_i una versione precedente (eventualmente coincidente con S), ed S_i^{AUG} il suo *schema aumentato*. Mentre la migrazione coinvolge i dati che hanno una validità che si sovrappone ad entrambe le versioni S ed S', l'aumento di S_i coinvolge solamente i dati che hanno la stessa validità temporale di S_i ($[t_1, t_2]$). A differenziare ulteriormente le due operazioni è la gestione degli eventi: mentre nella migrazione non vengono coinvolti, l'aumento deve prendere in considerazione tutti gli eventi che sono accaduti durante $[t_1, t_2]$ ed eventualmente alcune istanze di gerarchie non erano valide in $[t_1, t_2]$.

Come nella migrazione, tuttavia, esistono anche in questo caso delle azioni che non dipendono dalla specifica sequenza delle operazioni di modifica applicate, ma sull'effetto di rete per gli attributi e le dipendenze funzionali coinvolte nella transazione. In particolare:

- Se l'attributo A è stato aggiunto nel corso della transazione, cioè se $A \in \texttt{Diff}_\texttt{A}^+(S, S')$, allora il progettista può voler includere A nello schema aumentato per S_i per rendere possibili tutte le query cross-versionali che coinvolgono A.

- Se f è una nuova dipendenza funzionale, cioè $f = A \rightarrow B \in \texttt{Diff}_\texttt{F}^+(S, S')$, allora il progettista può voler includere f nello schema aumentato per S_i, per permettere i roll-up e drill-down cross-versionali che coinvolgono A e B.

Le operazioni di aumento degli schemi S_i coinvolgono solamente le operazioni di aggiunta, a differenza di quanto visto per la migrazione, per un motivo preciso. Mentre il progettista può aggiungere ed eliminare attributi e dipendenze funzionali dagli schemi, non è affatto utile ai fini delle interrogazioni cross-versionali la propagazione di una cancellazione negli schemi precedenti. Nell'ottica di trovare il sistema migliore per rendere possibile l'interrogazione sia degli schemi singoli che di schemi multipli, la cancellazione non farà parte dei nostri schemi aumentati.

Introduciamo quindi la definizione 22 di versione aumentata, costruita sulla base di S, S' e sullo schema aumentato basato su queste due versioni dello schema.

Definizione 22 (Versione Aumentata) *Sia $S_i^{AUG} = (\hat{U}_i, F_i)$ lo schema aumentato costruito sulla base delle operazioni di modifica dello schema S che portano allo schema S'. Siano $\widehat{\texttt{Diff}_\texttt{A}}(S, S')$ e $\widehat{\texttt{Diff}_\texttt{F}}(S, S')$ i sottoinsiemi di $\texttt{Diff}_\texttt{A}^+(S, S')$ e $\texttt{Diff}_\texttt{F}^+(S, S')$, rispettivamente, che includono solo gli attributi e le dipendenze funzionali che il progettista decide di inserire nelle operazioni di aumento. La* versione aumentata *di S_i^{AUG} è definita come* $\texttt{Aug}(S_i^{AUG}, S, S') := (\hat{U}_i \cup \widehat{\texttt{Diff}_\texttt{A}}(S, S'), (F_i \cup \pi_{\hat{U}_i \cup \widehat{\texttt{Diff}_\texttt{A}}(S,S')}(\widehat{\texttt{Diff}_\texttt{F}}(S, S')))^-)$

E' necessario che tutti gli attributi coinvolti dalle dipendenze funzionali presenti in $\widehat{\texttt{Diff}_\texttt{F}}(S, S')$ devono essere presenti in $\widehat{\texttt{Diff}_\texttt{A}}(S, S')$, dal momento che si possono aumentare solamente le dipendenze funzionali che coinvolgono attributi già presenti.

Le azioni di aumento associate a ciascun elemento in $\texttt{Diff}_\texttt{A}^+(S, S')$ e $\texttt{Diff}_\texttt{F}^+(S, S')$ sono riportate nella tabella 4.2 e sono definite come segue:

- *Stimare i valori per A*: Una nuova misura A è stata aggiunta. Per permettere le query cross-versionali, il progettista fornisce i valori per A per gli eventi

Tabella 4.2: Azioni di aumento associate agli inserimenti di attributi e dipendenze funzionali

Elemento	*Condizione*		*Azione di aumento*
$A \in \widehat{\texttt{Diff}_\texttt{A}}(S, S')$	$(E \to A) \in F_i$	A è una misura	stimare i valori per A
		A è una dimensione	disaggregare i valori delle misure
	$(E \to A) \notin F_i$	A è una misura derivata	calcolare i valori per A
		A è una proprietà	aggiungere i valori di A
$f \in \widehat{\texttt{Diff}_\texttt{F}}(S, S')$	-		controllare se f è soddisfatta

registrati in S_i, derivando oppure stimando i valori basandosi su quelli delle altre misure.

- *Disaggregare i valori delle misure*: Una nuova dimensione A è stata aggiunta. Per permettere le query cross-versionali, il progettista deve disaggregare gli eventi del passato attraverso A, in accordo con alcune regole di business oppure adottando una interpolazione statistica per dedurre la correlazione tra le misure [PS03].

- *Calcolare i valori per A*: Una misura derivata A è stata aggiunta. Per la definizione di misura derivata, i valori di A per il passato sono facilmente calcolabili sulla base di alcune operazioni su altre misure già presenti.

- *Aggiungere i valori per A*: Una proprietà A è stata aggiunta, quindi il progettista deve fornire i valori per A.

- *Controllare se f è soddisfatta*: Una nuova dipendenza funzionale f è stata aggiunta. Come per la migrazione, dobbiamo distinguere due casi:

  1. Se f coinvolge due attributi già esistenti, è necessario controllare se f è soddisfatta anche per S_i
  2. Se f coinvolge un attributo appena aggiunto A, allora i valori di A devono essere forniti in maniera tale che f sia soddisfatta.

4.4 Storia delle versioni

Definizione 23 (Storia delle Versioni) *Definiamo con* storia delle versioni *(*history*) la sequenza di una o più triple nella forma* (S_i, S_i^{AUG}, t_i)*, dove* S_i *è una versione,* S_i^{AUG} *lo schema aumentato connesso ad* S_i *e* t *è l'istante iniziale dell'intervallo di validità di* S*:*

$$H = (\ (S_0, S_0^{AUG}, t_0), \ldots, (S_n, S_n^{AUG}, t_n)\ )$$

dove $n \geq 0$ *e* $t_{i-1} < t_i$ *per* $1 \leq i \leq n$*. Bisogna notare che in ciascuna history per l'ultima tripla* (S_n, S_n^{AUG}, t_n) *abbiamo* $S_n^{AUG} = S_n$*, dato che l'aumento arricchisce solamente le versioni precedenti, utilizzando la conoscenza aggiunta nelle modifiche più recenti.*

Data la versione S_0 creata nell'istante t_0, la history iniziale è

$$H = ((S_0, S_0^{AUG}, t_0)),$$

dove $S_0^{AUG} = S_0$. Le modifiche allo schema successivamente cambiano la history nella maniera seguente: Sia $H = ((S_0, S_0^{AUG}, t_0), \ldots, (S_{n-1}, S_{n-1}^{AUG}, t_{n-1}), (S_n, S_n^{AUG}, t_n))$ una history, e S_{n+1} la nuova versione all'istante $t_{n+1} > t_n$; allora la history risultate H' sarà

$$\begin{aligned} H' = (\ & (S_0, \texttt{Aug}(S_0^{AUG}, S_n, S_{n+1}), t_0), \ldots, \\ & (S_n, \texttt{Aug}(S_n^{AUG}, S_n, S_{n+1}), t_n), \\ & (S_{n+1}, S_{n+1}^{AUG}, t_{n+1})\ ). \end{aligned}$$

dove $S_{n+1}^{AUG} := S_{n+1}$.

Ogni nuova modifica allo schema corrente può potenzialmente cambiare uno o tutti gli schemi aumentati contenuti nella history. Ogni aggiunta può portare alla propagazione all'indietro in tutti gli schemi aumentati. In aggiunta a questo, bisogna notare che i nuovi aumenti degli schemi precedenti sono basati sugli schemi aumentati registrati in precedenza, non sugli schemi stessi. In questa maniera le modifiche sono accumulate nel tempo, con il risultato che gli schemi aumentati raccoglieranno le informazioni in maniera sempre crescente.

4.5 Interrogazioni su schemi multipli

L'approccio al versioning presentato in questo lavoro è incentrato sulla possibilità di eseguire interrogazioni su schemi multipli, avvalendosi della disponibilità degli schemi aumentati e della history delle versioni. A differenza di quanto visto in [EK01], nel nostro approccio non è possibile interrogare tutto l'insieme degli eventi attraverso una unica versione. Ogni versione ha i suoi eventi corrispondenti, e solo attraverso di essa si può accedere ai dati relativi allo stesso periodo temporale di validità. Sotto questa limitazione, la possibilità di fare interrogazioni su schemi multipli è quindi subordinata all'individuazione di uno schema che funga da interfaccia verso tutte le versioni coinvolte. Questo schema deve inoltre essere univocamente determinato a partire dall'intervallo temporale di validità, attraverso una funzione di intersezione degli schemi. Definiamo quindi l'*operatore di intersezione*, denotato da $\otimes$, che permette di determinare lo schema comune di due versioni differenti, e per composizione di stabilire quindi lo schema comune di versioni multiple.

Definizione 24 *Sia* $S = (\{E\} \cup U, F)$ *e* $S' = (\{E\} \cup U', F')$. *Allora l'*intersezione di S con S', *denotata da* $S \otimes S'$, *è lo schema definito come*

$$S \otimes S' = (\{E\} \cup (U \cap U'), (F^* \cap F'^*)^-).$$

In maniera intuitiva, l'intersezione tra due versioni S ed S' è lo schema sul quale i dati registrati sotto S ed S' possono essere interrogati in maniera uniforme. Lo schema intersezione include quindi solo gli attributi che appartengono sia ad S che ad S', come pure per le dipendenze funzionali comuni. Un esempio dell'intersezione è riportata nella figura 4.9.

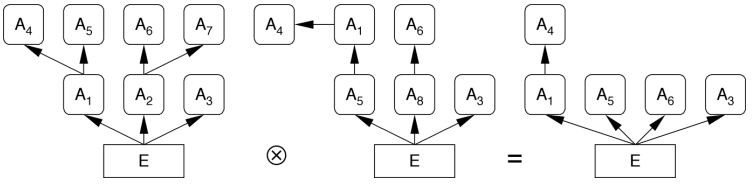


Figura 4.9: Intersezione tra due schema graph

Nella figura 4.10 viene mostrato l'effetto di una query Q_1 su un intervallo temporale $[t_1, t_3]$ che è un sottoinsieme di $[t_0, t_2]$, che a sua volta è l'unione degli intervalli

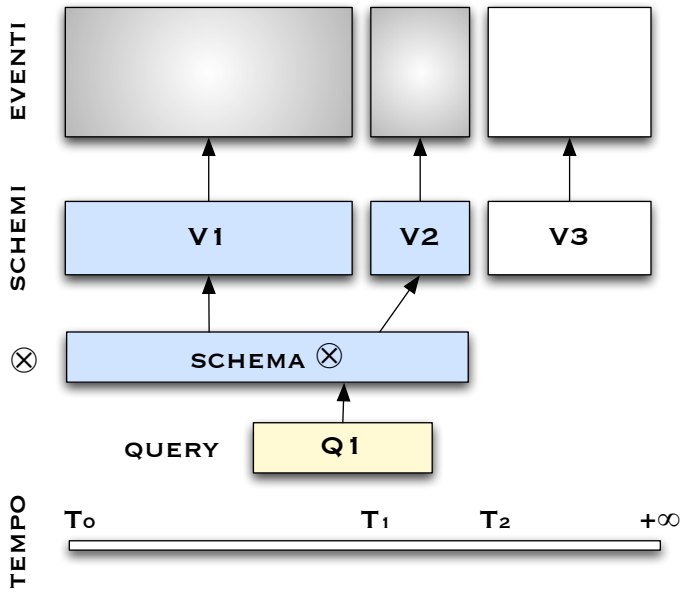


Figura 4.10: Interrogazione In Presenza Di Schemi Multipli

delle versioni V_1 e V_2. La query viene quindi applicata sullo schema comune di intersezione tra V_1 e V_2, individuato da $[t_1, t_3]$, con $t_0 < t_1 < t_2 < t_3$. Q_1 verrà automaticamente spezzata dal sistema di gestione delle query multi-schema in modo tale che il risultato sia l'unione dei risultati applicati a V_1 ed a V_2, in maniera trasparente per l'utente. Gli attributi a disposizione per la sessione OLAP saranno quelli dello schema intersezione, mentre gli eventi saranno quelli avvenuti nell'intervallo $[t_1, t_3]$.

L'operatore di intersezione è chiuso sugli gli schema graph, è commutativo ed associativo. Questo è molto importante, dal momento che ci permette di applicare l'operatore $\otimes$ ad *insiemi* di schemi.

Definizione 25 *Data una history H ed un intervallo temporale T non necessariamente contiguo, chiamiamo intervallo di T su H ($\texttt{Span}(\texttt{H},\texttt{T})$) l'insieme*

$$\texttt{Span}(H,T) = \{S_i^{AUG} \mid (S_i, S_i^{AUG}, t_i) \in H \ \wedge \ [t_i, t_{i+1}[\cap T \neq \emptyset\}$$

(assumendo per convenzione $t_{n+1} = +\infty$).

Lo schema comune *su H attraverso T è definito come* $\texttt{Com}(H,T) = \bigotimes_{\texttt{Span}(H,T)} S_i^{AUG}$.

L'esempio che segue mostra l'esecuzione di una query sia in presenza di operazioni di aumento sia in loro assenza.

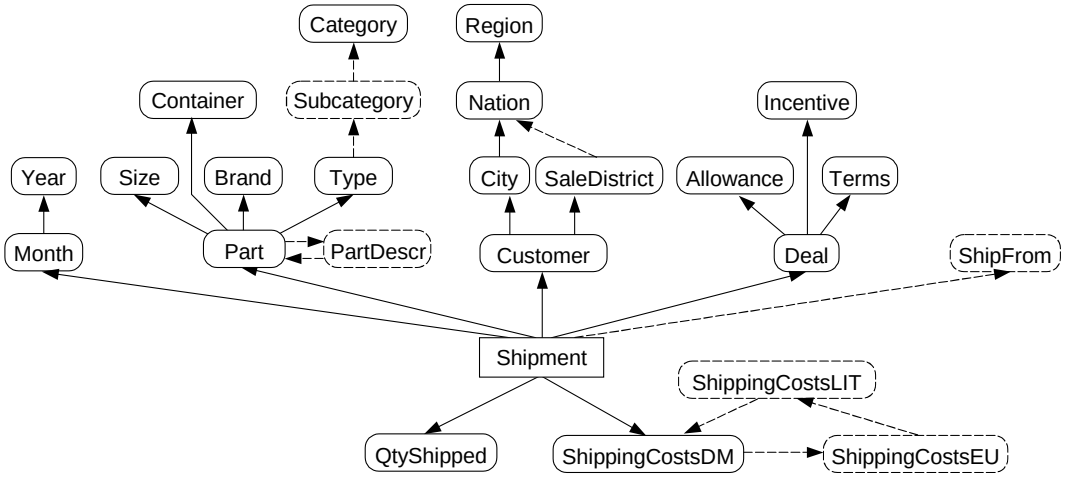


Figura 4.11: Le differenze sul contesto di formulazione di una interrogazione in assenza o presenza di operazioni di aumento

Esempio 7 *Sia* $H = ((S_0, S_0^{AUG}, t_0), (S_1, S_1^{AUG}, t_1), (S_2, S_2^{AUG}, t_2))$ *una history per il fatto Shipment (ricordando dagli esempi precedenti che* $t_1 = 1/1/2003$ *e* $t_2 = 1/1/2004$*), e sia* $q =$ "Calcola la quantità totale di ciascuna categoria di componenti spedita da ciascun deposito a ciascuna nazione dei clienti dal Luglio 2002". *L'intervallo temporale di* q *è* $T = [7/1/2002, +\infty[$*, dal momento che* $\texttt{Span}(H, T) = \{S_0, S_1, S_2\}$ *La figura 4.11 mostra il contesto della formulazione, definito da* $S_0^{AUG} \otimes S_1^{AUG} \otimes S_2^{AUG}$*, in due situazioni: nel caso in cui non sia stata effettuata nessuna operazione di aumento (descritto dalle linee semplici), e nel caso siano state fatte tutte le operazioni di aumento possibili (descritte dalle linee tratteggiate).*

Prima di tutto dobbiamo osservare che q è corretta solamente se ***ShipFrom*** *è stato inserito grazie ad una operazione di aumento in entrambe le versioni precedenti, dal momento che in sua assenza uno degli attributi richiesti non apparterrebbe al contesto della formulazione della query.*

Quindi osserviamo che, per esempio, le operazioni di drill-down da ***Category*** *a* ***SubCategory*** *saranno possibili solamente se le sotto-categorie e le loro relazioni di dipendenza con le categorie sono state stabilite anche per i dati del 2002. Il drill-down da* ***Nation*** *a* ***SaleDistrict*** *sarà possibile solo se la validità della dipendenza funzionale dai distretti di vendita alle nazioni è stata verificata anche per i dati inseriti prima del 2003.*

Infine dobbiamo notare che se ***ShippingCostsEU*** *viene aumentato potrà essere*

considerato su tutto il periodo di tempo, sebbene i costi di spedizione sono stati registrati solamente in Marchi tedeschi fino a t_2. Più in dettaglio, l'aumento delle misure derivate può essere implementato senza nessun costo di immagazzinamento, dato che può essere registrato come una semplice vista applicata con un fattore di conversione costante.

Nell'eseguire query che coinvolgono versioni differenti, sulla base delle scelte implementative adottate sarà necessario riscrivere la formulazione dell'utente per adattarla ai vari depositi di dati. Le query, quindi, dovranno essere interpretate e riscritte per adattarsi ai cambi di granularità delle dimensioni ed applicarsi ai repository relazionali sui quali sarà appoggiato il sistema.

Alcune informazioni dovranno essere presenti nei meta-dati, per permettere al sistema di eseguire la query in maniera univoca, mentre altre potranno essere derivate automaticamente dai dati già in possesso del sistema di interrogazione.

Capitolo 5

Un prototipo per la gestione del versioning

Una volta formalizzato l'approccio al versioning basato sugli schema graph e gli schemi aumentati, è necessario progettare un prototipo che permetta di dimostrare la validità teorica del procedimento, analizzarne eventuali necessità ed approfondire alcune scelte architetturali e progettuali. Questo capitolo è dedicato alla presentazione del prototipo per la gestione del versioning, all'analisi dei requisiti e delle scelte progettuali affrontate per realizzare un esempio concreto dell'applicazione del modello formale.

Il prototipo, che abbiamo chiamato DWers, dovrà gestire la creazione degli schema graph, la gestione della popolazione degli schemi aumentati, il controllo sulla history degli schemi e la manipolazione dei meta-dati necessari per l'utilizzo in un contesto reale di modellazione e di interrogazione ROLAP.

5.1 Architettura

Data la necessità di manipolazione degli schema graph, che sono basati su rappresentazioni di grafi orientati ciclici o aciclici, l'architettura alla base del prototipo deve necessariamente prevedere una interfaccia grafica per l'interazione del progettista del data warehouse e dell'utente finale che dovrà interrogarne il contenuto. In fase di progettazione si è optato per un applicativo unico, che permetta sia la modellazione dei grafi, sia il loro utilizzo ai fini delle interrogazioni ROLAP. Il prototipo lavorerà

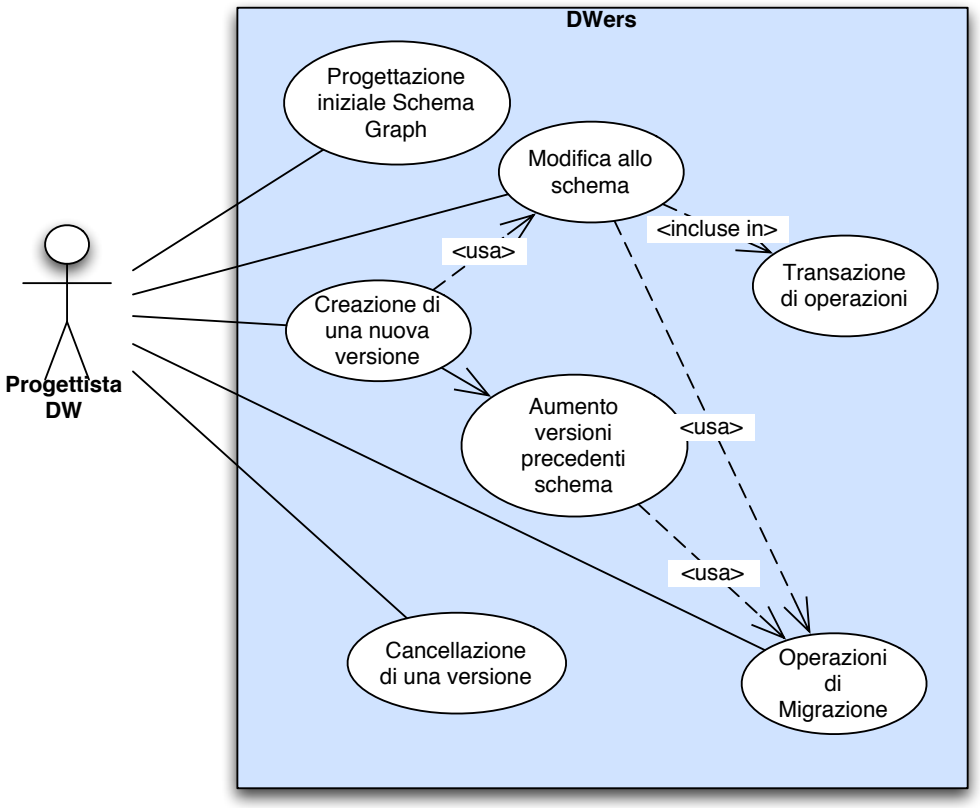


Figura 5.1: Diagramma dei casi d'uso del progettista del data warehouse

sui meta-dati delle history delle versioni degli schemi, appoggiandosi ad un dbms server. Pur trattandosi di un prototipo, uno dei requisiti che ci siamo posti in fase di progettazione è stata la ricerca della maggiore indipendenza dalle possibili architetture di sistema, affinché questo si potesse ben integrare con il numero maggiore di implementazioni già esistenti.

La figura 5.1 mostra un diagramma UML dei casi d'uso del progettista di data warehouse. Come abbiamo già visto nel capitolo precedente, l'operazione iniziale del progettista sarà quella di creare una history ed una versione iniziale. Attraverso una serie di operazioni di modifica alla versione corrente verranno aggiunte o eliminate delle dipendenze funzionali (ADD_F, DEL_F), oppure inseriti o cancellati degli attributi (ADD_A, DEL_A). La serie di operazioni elementari di modifica andrà a comporre una transazione, che il progettista deciderà di applicare una volta terminata questa fase. L'applicazione di una transazione di modifiche farà scattare la richiesta di aumento, ed al progettista verrà richiesto se arricchire con le informa-

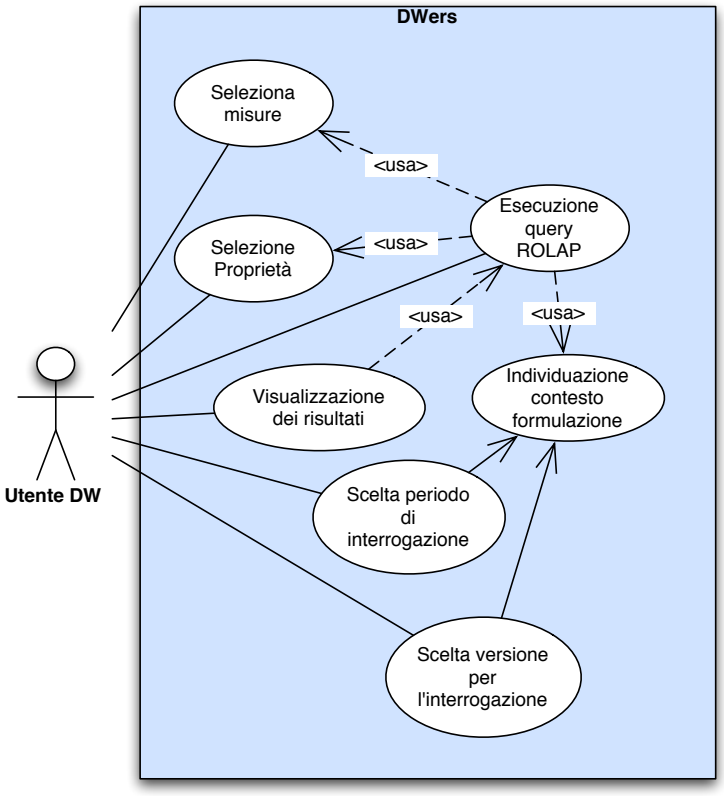


Figura 5.2: Diagramma dei casi d'uso dell'utente del data warehouse

zioni appena aggiunte le versioni precedenti a quella attuale. Nel caso il progettista decida di eseguire l'aumento, per ogni operazione elementare deve essere gestito il controllo o la migrazione dei dati corrispondenti. Come ultimo caso d'uso del progettista, bisogna anche considerare la cancellazione di una versione, che può avvenire in caso di inutilità o di estrema obsolescenza di uno schema.

La figura 5.2 mostra invece il diagramma UML dei casi d'uso dell'utente, che utilizzerà il sistema solamente per la visualizzazione degli schema graph e per l'interrogazione del data warehouse sulla base delle versioni. L'esecuzione di una query ROLAP inizia con la selezione del contesto della formulazione della query, individuando una specifica versione oppure un periodo temporale. L'utente dovrà selezionare le misure i cui risultati dovranno essere mostrati e le proprietà desiderate, eventualmente con clausole di proiezione per restringere il campo della selezione. Una volta eseguita la query potrà vederne i risultati e procedere con nuove interrogazioni.

La figura 5.3 mostra un tipico esempio dell'utilizzo di questo prototipo, i cui

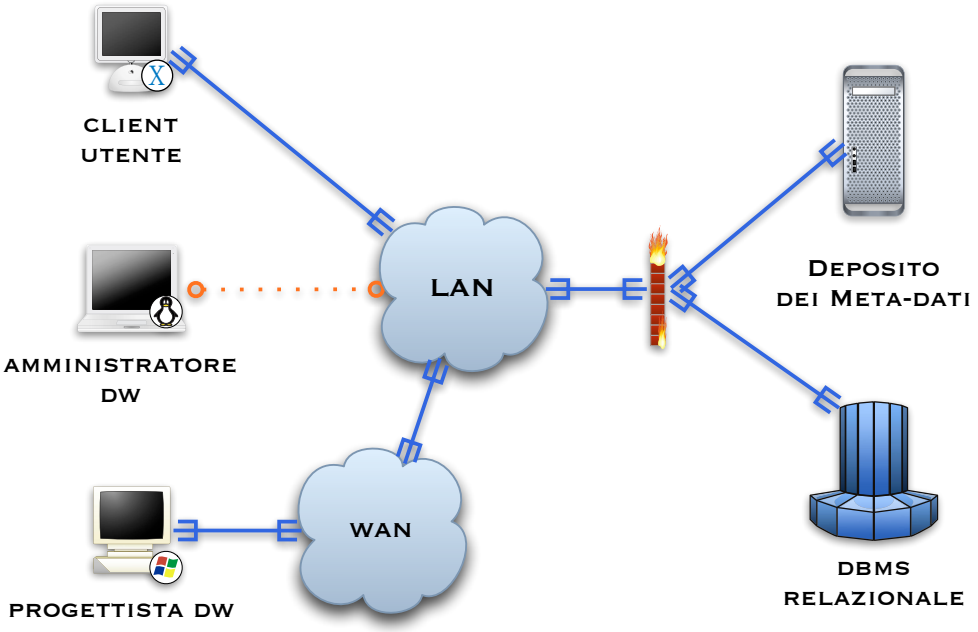


Figura 5.3: Architettura di rete del prototipo di versioning

componenti sono distribuiti all'interno di una LAN o WAN. L'applicazione DWers può risiedere nei pc client del progettista e degli utenti, indipendentemente dal sistema operativo utilizzato, i meta-dati possono essere salvati in locale oppure messi a disposizione attraverso un repository rsync, ftp o SMB, mentre il dbms server fornirà i dati estensionali necessari per le interrogazioni. Nell'esempio è presente anche una architettura di firewalling, per incrementare il livello di sicurezza dei dati contenuti nel DBMS. In fase di progettazione e modellazione del data warehouse, non sarà necessario avere a disposizione la connessione con il DBMS server. L'analisi dell'architettura di rete a supporto del sistema di gestione del versioning esula dal contesto di questo lavoro di tesi, quindi daremo una serie di assunti ed ignoreremo le problematiche di sicurezza e di connettività:

- Il prototipo deve risiedere sul computer dei client (Utente, Progettista)
- I meta-dati saranno contenuti in semplici file, che possono essere distribuiti attraverso la rete attraverso qualsiasi protocollo di scambio (ad esempio ftp, rsync, NFS, SBM, etc.)
- Il prototipo si connetterà al database relazionale attraverso il linguaggio rela-

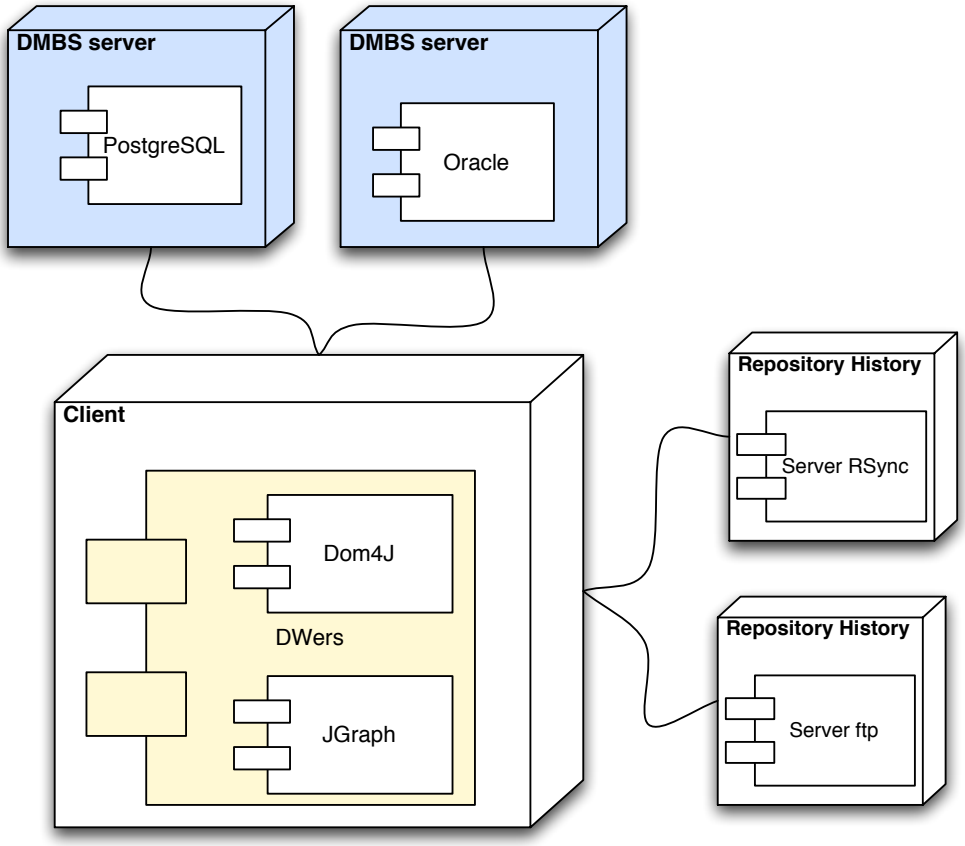


Figura 5.4: Diagramma UML dei componenti del prototipo

zionale SQL standard e driver di astrazione (ODBC, JDBC) dell'implementazione del DBMS, per permettere l'indipendenza dal tipo di server.

- I diversi livelli dell'architettura di rete del prototipo possono essere eventualmente contenuti all'interno di uno stesso hardware, oppure distribuiti in reti distribuite, in maniera trasparente per l'utente.

La figura 5.4 mostra chiaramente attraverso un diagramma UML dei componenti la distribuzione e le connessioni tra i diversi elementi che compongono l'architettura. In fase di implementazione del prototipo sono state fatte delle scelte architetturali e sono state individuate alcune componenti software che abbiamo deciso di utilizzare per la realizzazione di DWers. Dati i requisiti di indipendenza dall'architettura dei client e la necessità di utilizzare dei driver standard per la connessione con il database relazionale, la scelta del linguaggio di programmazione è ricaduta immediatamente su Java. Per quanto riguarda il formato dei meta-dati, è stato scelto

il linguaggio XML, data la sua estrema flessibilità e potenzialità di manipolazione ed interrogazione, come vedremo nella sezione successiva di questo capitolo. Per la realizzazione sono state utilizzate inoltre queste componenti software, già disponibili sotto licenza libera in rete:

- **DBMS Server PostgreSQL 8**: Questo prodotto è distribuito con licenza General Public License (GPL) ed è un DBMS solido e potente, spesso utilizzato come alternativa ad Oracle. Supporta le transazioni, le stored procedure, i trigger e tutte le funzionalità di cui avevamo necessità per la realizzazione del prototipo. Data la sua grande diffusione, esistono un buon numero di strumenti per la modellazione delle relazioni, per l'interrogazione delle basi di dati e per l'uso attraverso tutti i principali linguaggi di programmazione. Su Java, in particolare, esistono diversi driver jdbc molto validi.

- **Dom4J**: Questo framework XML java permette la creazione, l'interrogazione e la manipolazione di file XML, supportando completamente gli standard DOM e SAX e fornendo delle valide implementazioni degli strumenti XPath, XSLT ed XML Query.

- **JGraph**: Questo framework java permette la gestione e la manipolazione di grafi orientati in java.

- **JGraphaddons**: Questa estensione del framework JGraph permette di migliorare la gestione delle viste dei grafi, fornendo classi e strumenti che permettono ad esempio di ordinare i nodi attraverso algoritmi flessibili e personalizzabili

In seguito vedremo alcuni degli aspetti progettuali relativi a queste componenti, ma prima è necessario approfondire le necessità di elaborazione riguardanti i metadati relativi alle history delle versioni per il data warehouse, punto di partenza della progettazione del prototipo.

5.2 Meta-conoscenza

Come abbiamo già visto nel capitolo precedente, gli *schema graph* hanno bisogno di essere arricchiti con alcune informazioni fondamentali per la connessione con il

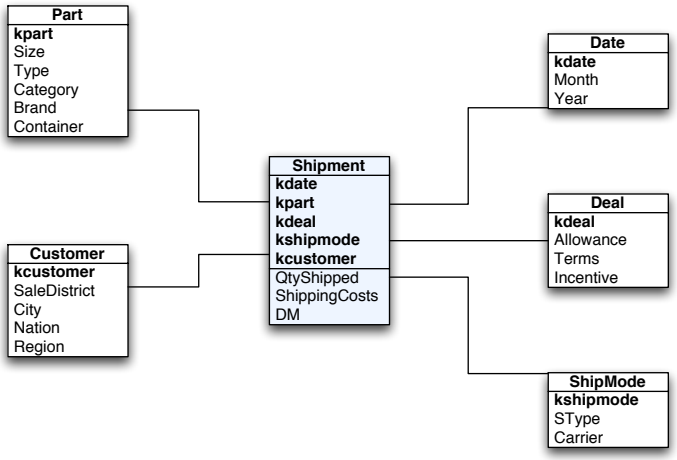


Figura 5.5: Schema a Stella del Fatto Shipment

database relazionale che contiene i dati del nostro data warehouse. Questo passaggio è di fondamentale importanza per comprendere le necessità di elaborazione reale del prototipo, che oltre a gestire la modellazione degli schema graph dovrà permettere la relazione tra il modello introdotto ed i dati estensionali. I nostri meta-dati, quindi, dovranno contenere sia tutte le informazioni necessarie alla costruzione degli *schema graph*, sia tutte le informazioni aggiuntive necessarie alla ricostruzione di uno schema logico per l'inserimento ed il recupero dei dati all'interno del DBMS relazionale.

Schema a Stella Per la costruzione del nostro prototipo, abbiamo utilizzato il modello logico dello *schema a stella*, come definito in [Gro96, VS99]. Lo schema a stella del fatto shipment utilizzato in questo lavoro di tesi come esempio è mostrato nella figura 5.5. Come si può vedere anche nella figura, a livello logico occorre distinguere tra misure, dimensioni e proprietà, presenti rispettivamente nella *fact table* e nelle *dimension table*. Vedremo in seguito come questo verrà effettuato all'interno della definizione del nostro schema dei meta-dati.

Per la memorizzazione dei meta-dati abbiamo scelto dei file XML, definendone la struttura attraverso uno XML Schema, standard del World Wide Web Consortium [W3C01].

Una differenza sostanziale tra il modello utilizzato per la memorizzazione dei dati ed il concetto da modellare risiede nella struttura: mentre il contenuto dei file XML ha una gerarchia ad albero, gli schema graph sono grafi che possono contenere cicli e gerarchie multiple. Questo, quindi, ci impedisce di utilizzare direttamente

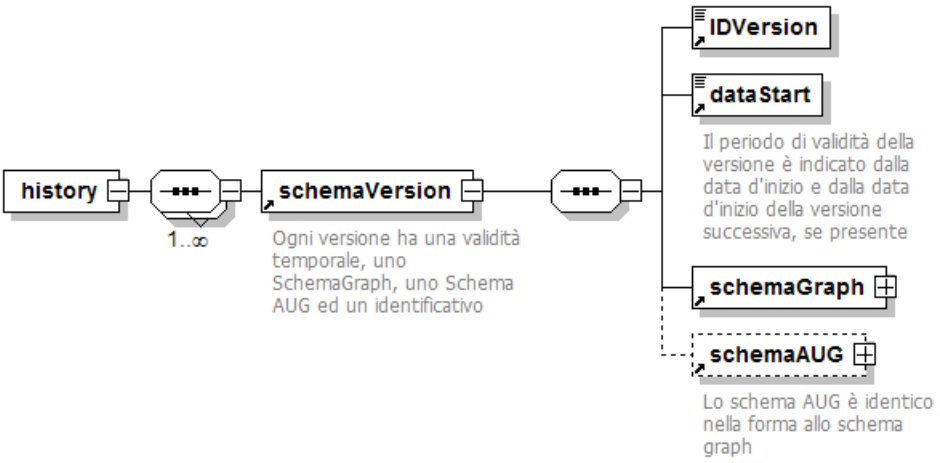


Figura 5.6: W3C XML Schema della History

la gerarchia degli elementi XML per la memorizzazione delle dipendenze funzionali, che andranno modellate in elementi a parte attraverso dei riferimenti agli attributi. Altre relazioni, come l'appartenenza di un attributo ad uno schema, possono invece essere modellate attraverso i tag XML e la gerarchia ad albero, come vedremo in seguito.

Per la descrizione della struttura dei file che contengono i meta-dati, partiremo dal concetto più generale per arrivare al dettaglio, in maniera inversa rispetto alla definizione formale. Questo sia per facilitare la lettura, sia per descrivere le componenti in coerenza con la struttura gerarchica ad albero del file XML.

History

Le history degli schemi, come abbiamo definito nel paragrafo 4.4 nella `DEF 23`, sono serie di triple (S_i, S_i^{AUG}, t_i), dove S_i è una versione, S_i^{AUG} lo schema aumentato connesso ad S_i e t è l'istante iniziale dell'intervallo di validità di S.

Ogni history è memorizzata in un file di meta-dati proprio, quindi i prototipo DWers memorizzerà un file differente per ogni storia degli schemi.

Come mostrato nella figura 5.6, le history sono composte da una sequenza potenzialmente illimitata di versioni (`schemaVersion`). Le versioni sono le triple definite in precedenza, composte da uno schema graph ($S_i \rightarrow$ `schemaGraph`), un istante iniziale di validità ($t_i \rightarrow$ `dataStart`), una etichetta ($i \rightarrow$ `IDVersion`) ed uno schema

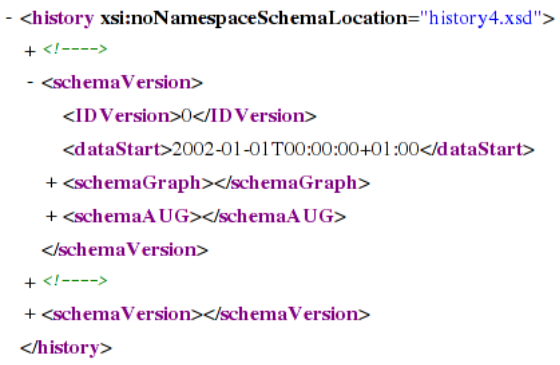


Figura 5.7: Esempio della struttura XML di una History

aumentato opzionale ($S_i^{AUG} \rightarrow$ `schemaAUG`). La cardinalità minima degli schemi aumentati all'interno di una versione è 0, perché la sua presenza non è necessaria in almeno due casi:

- La versione i è l'ultima definita. Lo schema aumentato coincide pertanto con lo schema graph della stessa versione ($S_i^{AUG} \equiv S_i$).
- Il progettista del versioning del data warehouse non ha ritenuto necessario aumentare lo schema graph S_i con le modifiche apportate alle nuove versioni che sono state create successivamente. Anche in questo caso lo schema aumentato coincide con lo schema graph della stessa versione ($S_i^{AUG} \equiv S_i$) e non si ritiene utile replicare la porzione dei meta-dati relativa. Quando almeno una operazione di modifica verrà aumentata nello schema S_i si verrà a creare un S_i^{AUG}

Le versioni vengono memorizzate sotto l'elemento `schemaVersion`, ed hanno una validità temporale che è data dall'intervallo tra l'istante iniziale t_i (`dataStart`) e l'istante iniziale t_{i+1} se $\exists(i+1) \rightarrow (S_{i+1}, S_{i+1}^{AUG}, t_{i+1}) \in H$

Schema graph

Gli schema graph sono il punto centrale della meta-conoscenza legata alle history delle versioni. Come mostrato nella figura 5.8, la loro composizione è abbastanza complessa ma perfettamente riconducibile alla definizione data in precedenza. Ogni schemaGraph è formato da un solo *fact node* (`factNode`) e da una sequenza potenzialmente infinita di misure, proprietà e dipendenze funzionali.

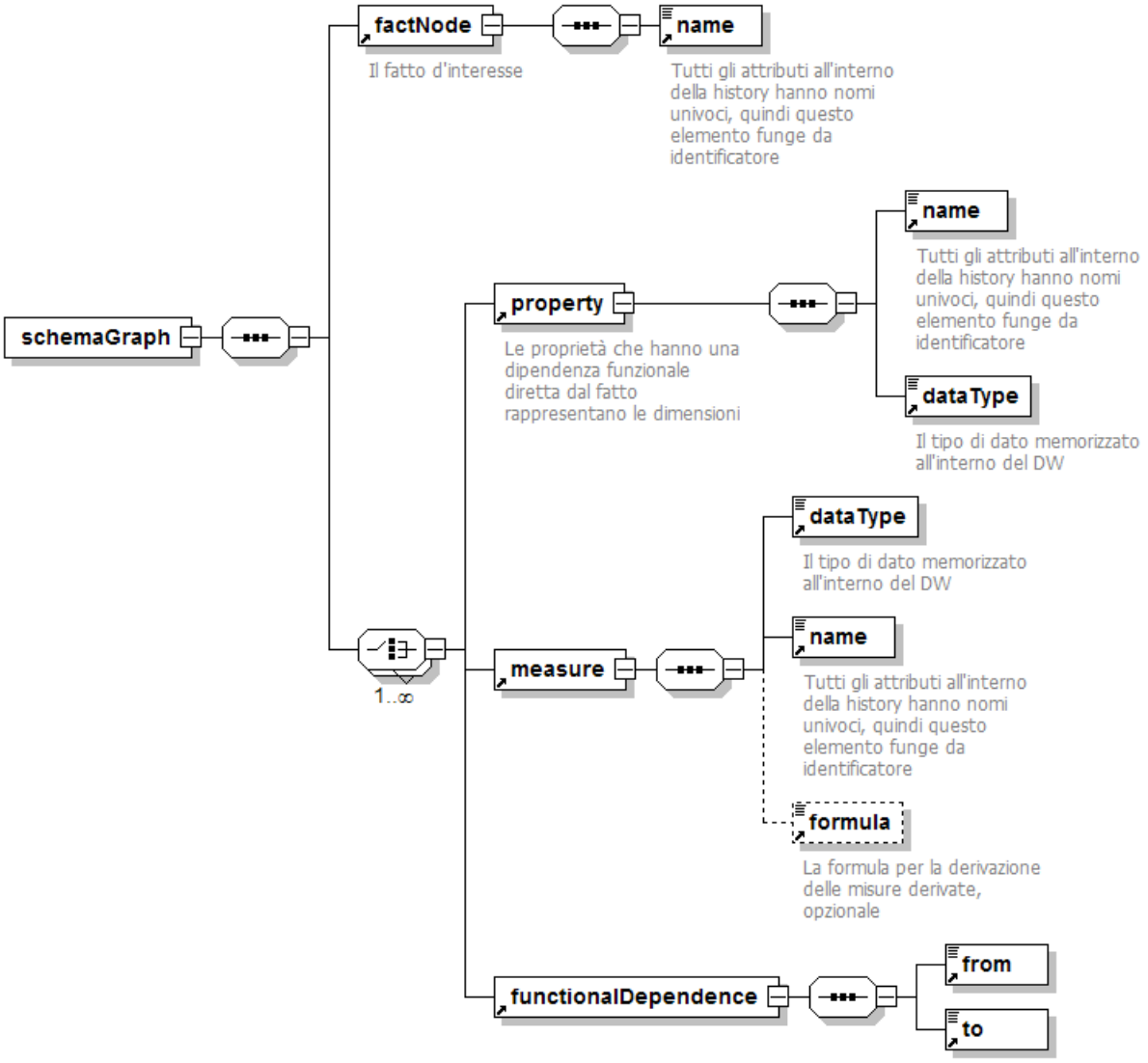


Figura 5.8: W3C XML Schema dello Schema Graph

All'interno di una history il fatto d'interesse non può cambiare, ciò nonostante si è deciso di salvare il fact node in ogni schema graph ed ogni schema aumentato. Questa scelta è dovuta all'utilizzo delle query XPath all'interno dell'applicativo per la gestione delle dipendenze funzionali. Questa ridondanza viene controllata a livello implementativo nel codice di DWers, e non crea problemi dal momento che non viene mai permessa nessun modifica al nodo del fatto.

E' importante notare come vengono gestiti gli attributi all'interno degli schema graph, perché non è stato definito nessun elemento generico che li rappresenta.

Il rapporto con lo schema logico dello schema graph dipende dalle scelte implementative fatte sul database relazionale, i cui dati per la connessione andranno inseriti in un file a parte oppure attraverso una estensione del XML Schema qui presentato. In ogni caso, se è possibile scegliere il nome delle relazioni in maniera autonoma, sia che si lavori su un repository single-pool sia che si scelga un multi-pool sarà possibile determinare in maniera univoca le tabelle coinvolte. In caso contrario, sarà necessario inserire altri dati che permettano di far corrispondere i nodi degli attributi al loro repository.

Attributi

Negli schema graph, così come sono stati presentati nella definizione 13 introdotta nel paragrafo 4.1.2, le proprietà e le misure derivate (a) non sono distinte, così come non lo sono le dimensioni e le misure (b). E' facile, a partire dallo schema graph, stabilire se un attributo appartiene alla categoria (a) oppure alla categoria (b):

> Sia $A \rightarrow B$ una dipendenza funzionale contenuta in F, l'insieme delle dipendenze funzionali di uno schema graph S, E il fact node ed F^- l'insieme delle dipendenze funzionali dello schema graph ridotto come definito nel paragrafo 4.1.3 in `DEF` 17. Se $(A \rightarrow B \in F^-) \wedge (A = E)$, allora B è una dimensione o una misura, altrimenti B è una proprietà o una misura derivata.

Dovendo connettere il nostro modello rappresentato dagli schema graph ad un modello logico, come ad esempio lo *schema a stella*, avremo quindi la difficoltà di definire le misure all'interno della *fact table* e le proprietà all'interno delle *dimension table*. Nelle figure 4.4, 4.5 e 4.6 la distinzione grafica tra misure e dimensioni è data

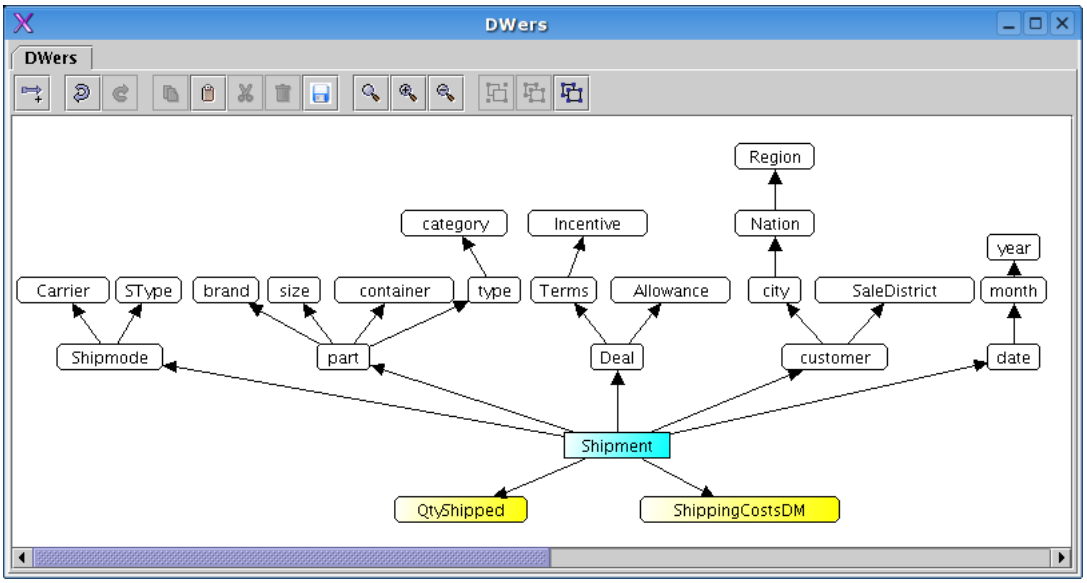


Figura 5.9: Schema Graph del fatto Shipment

dalla posizione degli attributi rispetto al *fact node*. Tutti i nodi inseriti sotto il *fact node* rispetto all'asse verticale rappresentano misure e misure derivate, mentre quelli sopra rappresentano le dimensioni e le proprietà.

Dal punto di vista formale, quindi, della definizione dello schema dei meta-dati, bisognerà distinguere tra proprietà e misure.

Come si vede quindi nello XML Schema relativo allo schema graph (figura 5.8), al momento del salvataggio il progettista dovrà avere distinto in maniera esplicita se un attributo è una misura oppure una proprietà.

In coerenza con la definizione di schema graph introdotta nel paragrafo 4.1.2, abbiamo suddiviso l'insieme U degli attributi in P, insieme delle proprietà e delle dimensioni dello schema, ed M, l'insieme delle misure (semplici e derivate):

$$U = P \cup M$$

La figura 5.9 mostra una schermata di DWers in fase di modellazione dello schema graph relativo al fatto `Shipment`. Come si può notare, fatto, misure e proprietà sono stati distinti tramite una diversa colorazione dei nodi che li rappresentano, rispettivamente in azzurro, giallo ed in bianco.

All'atto dell'inserimento di un nuovo attributo, come mostrato in figura 5.10, viene richiesto il nome ed il tipo di dato. Una volta inserito, l'attributo è connesso direttamente con una dipendenza funzionale al fact node, ed è una proprietà. In

Figura 5.10: Inserimento di un nuovo attributo

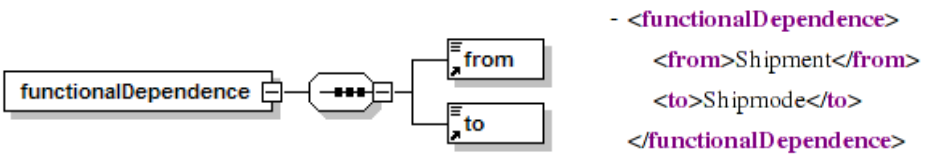
Figura 5.11: Modifica di un attributo da proprietà a misura

qualsiasi momento all'interno di una transazione il progettista può trasformare una gerarchia di attributi da proprietà a misure e viceversa.

Come mostrato in figura 5.11, questa operazione è molto semplice. Siccome non è possibile creare dipendenze funzionali tra un componente di M ed uno di P, l'eventuale connessione di due nodi di classe differente deve essere seguita dalla specificazione della classe di tutta la porzione di grafo corrispondente.

Le dimensioni e le misure derivate non sono state distinte rispettivamente dalle proprietà e dalle misure perché sono di facile derivazione rispetto all'insieme delle proprietà e delle dipendenze funzionali.

Sia $p \in P$. Se $\exists f \in F^-$ tale che $f = E \rightarrow p$ allora p è una dimensione, viceversa è una proprietà. Analogamente sia $m \in M$. Se $\exists f \in F^-$ tale che $f = E \rightarrow m$ allora m è una misura, viceversa è una misura derivata. Per quanto riguarda le misure derivate, questa distinzione serve solo per controllare che le misure non direttamente inserite nella fact table siano derivabili attraverso una formula di derivazione, che può produrre una vista che può essere materializzata, se la sua complessità computazionale è tale da doverne ridurre il peso in fase di interrogazione.



(a) (b)

Figura 5.12: Dipendenza Funzionale: struttura (a), esempio (b)

Fact Node, proprietà e misure sono quindi univocamente determinate dall'elemento figlio name, che come abbiamo già visto è univoco in tutta la history come richiesto dalla *Universal Relation Schema Assumption*.

L'elemento `dataType` è necessario in fase di creazione in un nuovo nodo, per permettere al sistema di creare automaticamente l'attributo nella relazione della dimension table nel caso si tratti di una proprietà, oppure nella fact table nel caso si tratti di una misura.

L'elemento opzionale `formula` serve nel caso si inserisca una misura derivata, che necessita della stringa SQL per la creazione della vista sulla misura semplice corrispondente.

Dipendenze funzionali

Le dipendenze funzionali prese in esame in questo lavoro di tesi sono semplici, quindi possono essere descritte come coppie di riferimenti (`from`, `to`) che determinano la gerarchia dei nodi del grafo, come mostrato nella figura 5.12.

Non sarà possibile quindi definire una dipendenza multipla, come indicato nella definizione formale di schema graph.

All'atto di creazione di una nuova dipendenza funzionale f, occorre fare alcune verifiche per evitare inconsistenze e per garantire che lo schema sia univocamente determinato:

1. La dipendenza funzionale f non deve essere già presente nello schema, sia in forma esplicita che implicita. Se la nuova dipendenza funzionale che si vuole creare non aggiunge nessuna conoscenza al grafo, allora il sistema deve impedirne l'inserimento.

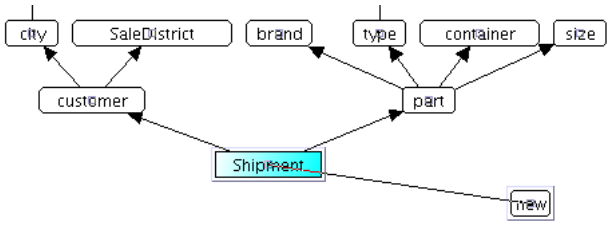


Figura 5.13: Creazione di una nuova dipendenza funzionale

2. Dato l'insieme delle precedenti dipendenze funzionali F_i, se la nuova dipendenza aggiunge conoscenza al grafo, è necessario determinare l'insieme delle dipendenze funzionali ridotto F_{i+1}^{-} di $F_{i+1} = \{f\} \cup F_i$. Questa operazione è sempre possibile, come visto nel capitolo precedente ed in [Lec04].

Le operazioni che derivano dalla (2) sono presenti nella quasi totalità dei casi, visto che tutti gli attributi hanno sempre almeno una dipendenza funzionale, ed appena inseriti sono direttamente connessi al fatto. Nella maggior parte dei casi, quindi, l'inserimento di f comporterà l'eliminazione di tutte le dipendenze funzionali implicitamente determinate da f e precedentemente presenti.

Aggiornamento degli schemi

I meta-dati corrispondenti agli schemi aumentati sono esattamente gli stessi degli schemi graph. Mentre i due grafi sono di fatto indipendenti a livello dei meta-dati, a livello fisico in presenza di uno schema aumentato non ci sarà una duplicazione dei dati estensionali, perché lo schema originale può essere calcolato come una proiezione su S^{AUG}. Questo accade perché la validità temporale di S_i ed S_i^{AUG} è la stessa, e quindi i dati corrispondenti alle interrogazioni su S_i sono una proiezione dei dati corrispondenti alle interrogazioni su S_i^{AUG}.

La figura 5.14 mostra la porzione dello XML Schema corrispondente allo schema aumentato.

5.3 Progettazione

Una volta definiti i requisiti di elaborazione dei meta-dati da parte del prototipo, è necessario procedere con la progettazione dell'applicativo, per arrivare al diagramma

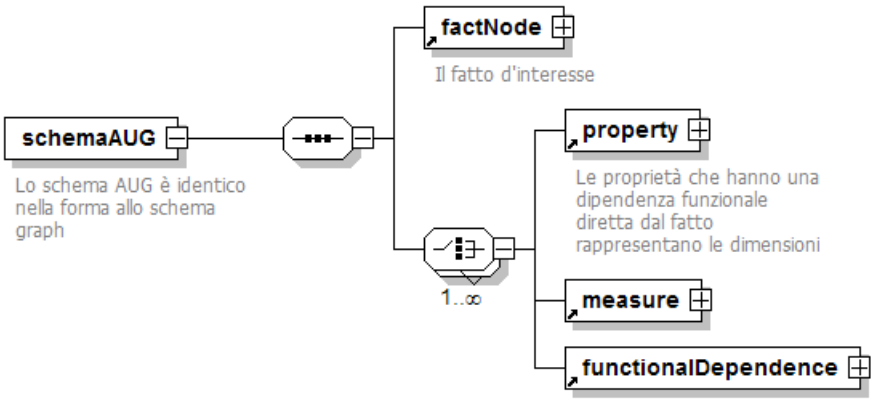


Figura 5.14: W3C XML Schema dello Schema Aumentato

delle classi ed alla successiva implementazione.

Nella figura 5.15 vediamo il diagramma UML delle classi risultante da questa fase di progettazione.

La classe principale che verrà eseguita è `DWers`, che contiene il metodo main per l'esecuzione dell'interfaccia grafica. Da questa classe sarà possibile aprire i file contenenti le history, inviate tramite argomenti a linea di comando (`args`), oppure aperti attraverso l'apposito menu. Le `history` hanno solamente due attributi: l'indirizzo del file di riferimento ed una struttura dati che contiene una lista delle versioni presenti nella history. Ogni history deve contenere almeno una versione dello schema (`schemaVersion`), che come abbiamo mostrato nella definizione dei meta-dati è composta da un identificativo di versione (`version`), il periodo di validità $[dataStart, dataStop]$, uno schema aumentato (`schemaAUG`) opzionale ed uno schema Graph (`schemaGraph`). Una differenza tra la progettazione della classe `schemaVersion` e la sua struttura di memorizzazione è l'intervallo di validità. Mentre nei meta-dati viene memorizzato solo l'istante iniziale dell'intervallo, qui sarà conveniente avere entrambi gli estremi. Questo è utile perché nella maggior parte del tempo di utilizzo del prototipo verrà mantenuta in memoria una versione alla volta, ed il recupero di quella successiva per l'individuazione del secondo estremo sarebbe dispendioso.

La classe `XMLData` è molto importante, ed è il cuore dell'elaborazione dei meta-dati. Viene istanziata al momento dell'apertura del file di elaborazione, e permette

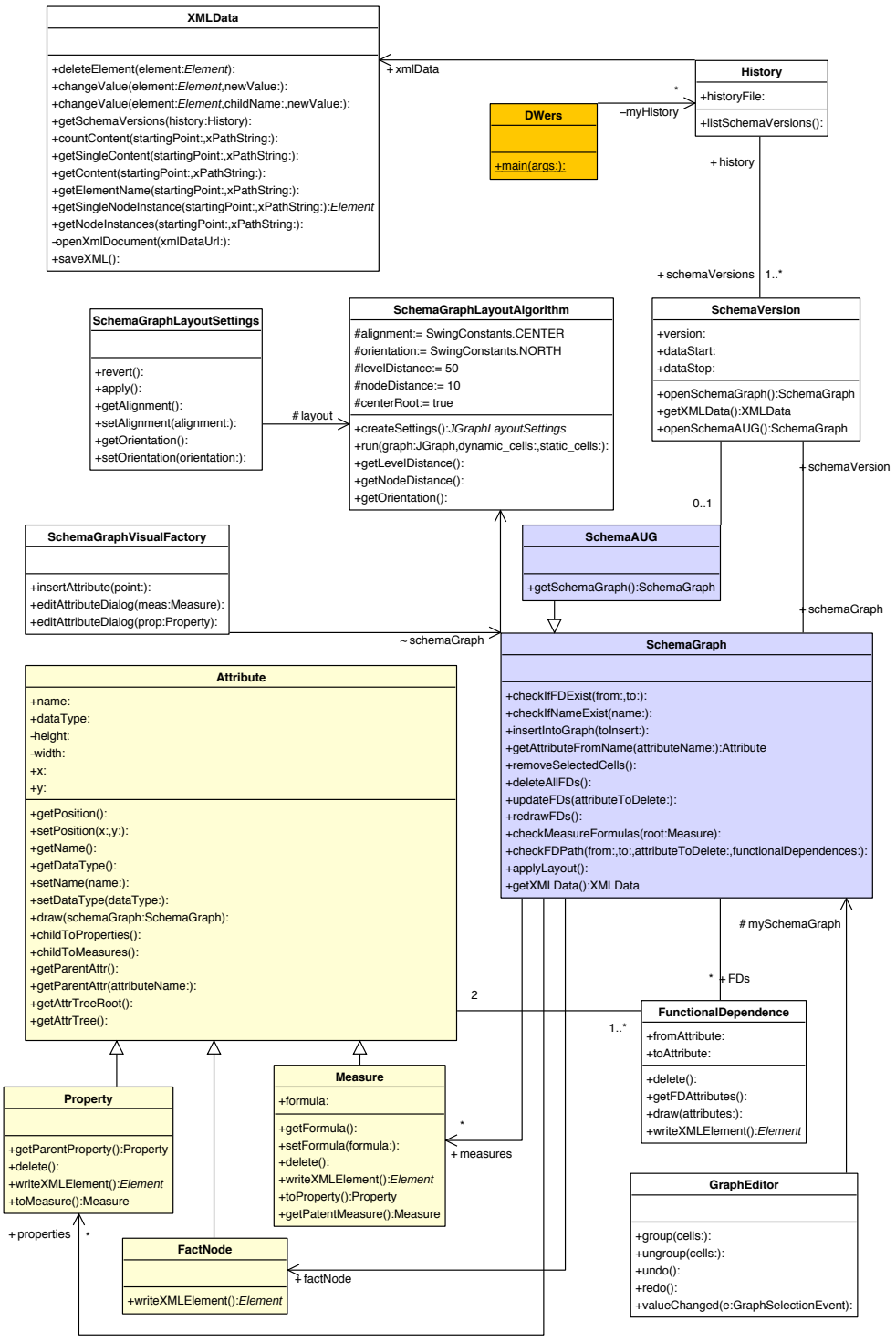


Figura 5.15: Diagramma UML delle Classi del prototipo DWers

di eseguire interrogazioni sulla meta-conoscenza dei grafi della history. Per ogni `history`, quindi, esiste una classe `XMLData` che carica l'albero della struttura XML e permette di gestire gli elementi a basso livello. Successivamente vedremo la gestione degli attributi nei vari livelli che li compongono: nodo del grafo, meta-dato ed istanza nel database. I metodi della classe `XMLData` possono essere suddivisi in tre categorie, in base alla funzione svolta:

- **Interrogazione dei meta-dati**: Contiene tutti i metodi per il recupero dei contenuti dell'elemento XML ed il recupero di puntatori all'elemento all'interno dell'albero XML. Il nome di questi metodi inizia per get.
- **Modifica del valore di un elemento**: Il metodo changeValue serve per modificare il contenuto di un elemento XML, sia nel caso di un elemento singolo che nel caso di un albero di elementi.
- **Cancellazione di un elemento**: Il metodo deleteElement elimina completamente un elemento, in seguito alla cancellazione di un attributo.

Per quanto riguarda la creazione di un nuovo elemento XML, invece, è necessario distinguere tra le varie tipologie di nodi (proprietà, misure, dipendenze funzionali), ed i metodi relativi a questa operazione sono inseriti nelle classi specifiche.

La classe `XMLData` fa largo uso delle API messe a disposizione dal framework Dom4J e dell'interfaccia di i/o di questa libreria. Per l'interrogazione dei meta-dati in fase di implementazione è stato utilizzato il linguaggio **XPath**. Questo potente strumento, assieme alla definizione della struttura attraverso l'XML Schema, ci permette di recuperare le informazioni salvate nel file XML. In aggiunta a questa possibilità, l'interrogazione dei meta-dati può avvenire anche durante una transazione, quando le nuove aggiunte ancora non sono state salvate nel file. Tutte le modifiche, infatti, vengono riportate nel `org.dom4j.Document` appena modellate, prima ancora della chiusura della transazione. In questo modo è possibile sempre trovare tutti le dipendenze funzionali relative ad un attributo, tutte le proprietà di una dimensione, etc. Tutte le modifiche apportate al `Document` da `writeXMLElement` e `deleteElement`, vengono salvate solamente con il metodo `saveXML`, ma sono disponibili per le interrogazioni anche prima di questa operazione. Le stringhe XPath per le interrogazioni sono gestite in maniera relativa. Questo significa che dovrò

salvare un riferimento `Element` (org.dom4j.Element) per ogni punto di interesse del file dei meta-dati, che mi permetterà di interrogare solo quella porzione della mia meta-conoscenza. Quando apro uno schema graph, ad esempio, salvo in memoria anche un puntatore all'Element nel documento XML.

Esempio 8 *Per selezionare tutte le proprietà contenute in uno schema graph istanziato da mySchemaGraph, mi basterà eseguire questa porzione di codice:*

mySchemaGraph.getXMLData().getNodeInstances(
mySchemaGraph.element, "property ");

Se invece volessi trovare tutte le proprietà che sono immediatamente precedenti ad una proprietà selezionata nella gerarchia delle granularità di una dimensione, avrei ad esempio:

schemaGraph.getXMLData().getContent(schemaGraph.element,
"functionalDependence/to[.=' " + attributeName + " ']//../from");

La classe `schemaGraph` rappresenta i grafi così come formalmente introdotti, e contiene i metodi necessari alla modifica ed alle elaborazione specifiche di questo elemento. Questa classe deve contenere sia le informazioni relative alla definizione dello schema, sia quelli relativi al suo disegno e modellazione su schermo.

Nella figura 5.16 possiamo vedere come la vista del grafo sia indipendente dal suo modello, che è definito univocamente. Due differenti viste del grafo possono avere i nodi disegnati o ordinati in maniera diversa, ma questo non incide sulla meta-conoscenza che modellano.
Per la visualizzazione del grafo e l'interfaccia del sistema di versioning in fase di implementazione si è deciso di utilizzare il framework JGraph.

La maggior parte dei metodi contenuti nella classe schemaGraph servono per controllare la consistenza del grafo prima o dopo l'inserimento o la cancellazione di un elemento:

- `checkIfFDExist`: controlla che una dipendenza funzionale non esista già all'interno del grafo, in maniera implicita o esplicita.

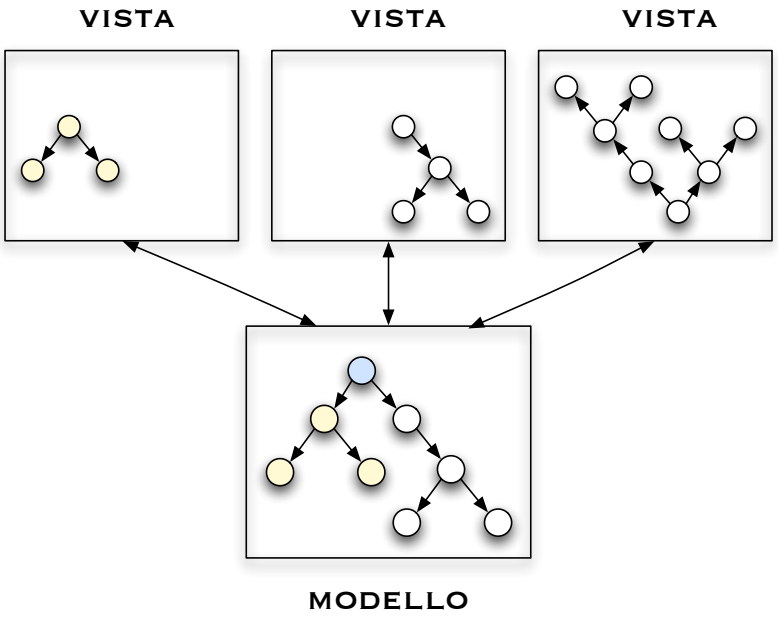


Figura 5.16: Rapporto tra modello del grafo e viste

- `checkIfNameExist`: controlla che non venga violata l'assunzione dell'univocità dei nomi degli attributi.
- `checkMeasureFormulas`: controlla che tutte le misure derivate abbiano una formula di derivazione rispetto ad una misura inserita nella fact table.
- `checkFDPath`: controlla l'esistenza di un percorso di dipendenze funzionali da un nodo ad un altro, utilizzato nei cicli di dipendenze funzionali.
- `updateFDs`: aggiorna le dipendenze funzionali dopo la cancellazione di un attributo.

Il metodo `getAttributeFromName` permette il recupero di un attributo a partire dal suo nome per l'eventuale modifica o cancellazione, `insertIntoGraph` inserisce un nuovo elemento o una dipendenza funzionale nella rappresentazione grafica, mentre `applyLayout` ridisegna il grafo secondo un algoritmo ottimale specificato in una classe a parte, per separare i dettagli implementativi della vista da quelli del modello.

La classe `SchemaGraphVisualFactory` fornisce le operazioni di modifica sul grafo che hanno bisogno di una interfaccia di supporto, come la modifica di un valore di un attributo oppure il cambiamento da proprietà a misura, o l'inserimento di un nodo

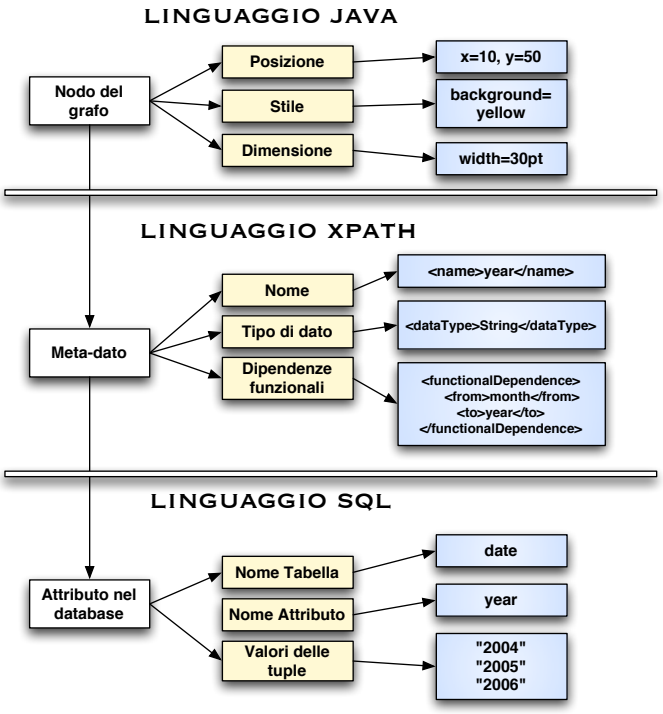


Figura 5.17: Un attributo visto dai tre livelli: grafo, meta-dati, database

in un punto. `GraphEditor` rappresenta invece una interfaccia grafica che permette le operazioni di undo, redo, di raggruppamento e di zoom del grafo, così come la raccolta degli eventi di modifica sul grafo.

La classe `schemaAUG` è una specificazione della classe schemaGraph, con l'aggiunta dei metodi per ricavare lo schema comune tra due versioni diverse. Come visto nel capitolo precedente, il risultato di questa operazione è determinato univocamente, e serve sia in fase di interrogazione dei dati sia in fase di modellazione delle versioni. Il fatto che `schemaGraph`, lo `schemaAUG` e lo schema comune abbiano praticamente gli stessi metodi e gli stessi attributi permette di gestire le operazioni di modellazione in maniera estremamente flessibile: il progettista può fare composizioni di schemi, fare modifiche ai risultati e salvarli, applicando le novità alle versioni precedenti, senza che questo comporti inconsistenze.

Prima di procedere con la descrizione delle classi che manipolano attributi e dipendenze funzionali, è necessario mostrare come si è deciso di suddividere la gestione dei livelli di informazione relativi a questi elementi.

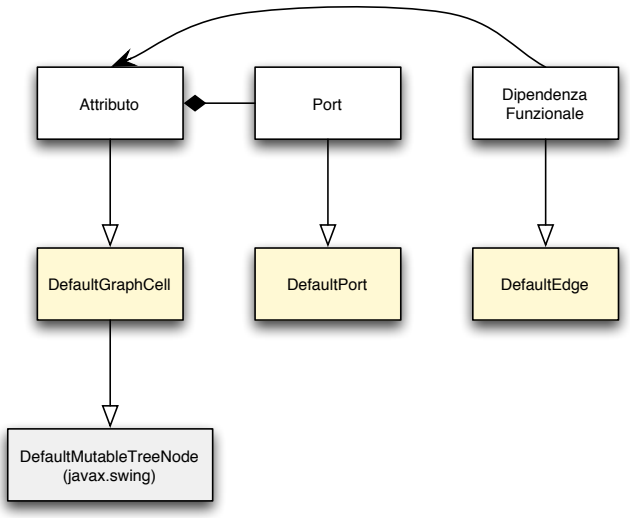


Figura 5.18: Un grafo in JGraph è composto da nodi (`GraphCell`), archi (`Egde`) e magneti (`Port`)

La figura 5.17 mostra i tre livelli che separano la gestione di un attributo d'esempio, la proprietà `year`. Il primo livello di dettaglio, rappresentato dalle informazioni grafiche del nodo preso in esame e dalla vista del grafo, è importante per l'utente ma di scarso interesse per le informazioni che modella. Per questo motivo le informazioni relative alla gestione grafica della rappresentazione dello schema non vengono mai salvate, e ad ogni sessione di utilizzo del prototipo vengono irrimediabilmente perse. A livello dei meta-dati, invece, è importante mantenere ogni dettaglio di conoscenza utile inserito dall'utente, da mettere a disposizione in ogni successiva sessione di lavoro. Per ultimo, ma non per ordine di importanza, nel livello del database vengono inseriti i dati relativi alle istanze dell'attributo modellato, così come tutte le informazioni necessarie in fase di interrogazione. Queste informazioni sono disponibili anche in maniera indipendente dai livelli precedenti, e sono il cuore della conoscenza acquisita nel corso degli anni, immagazzinata nel data warehouse.

La progettazione delle classi che gestiscono gli attributi e le dipendenze funzionali, pertanto, dovranno contenere metodi relativi ad ognuno di questi livelli, possibilmente in maniera separata per facilitare la portabilità delle scelte implementative.

Nel primo livello, dove l'implementazione è fortemente legata al framework JGraph, il grafo è composto da nodi (`GraphCell`), archi (`Egde`) e magneti (`Port`), come

mostrato in figura 5.18. Gli attributi, quindi, sono una estensione delle `GraphCell` definite, mentre le dipendenze funzionali sono una estensione degli archi del grafo (`Edge`). I magneti servono per fornire i punti di connessione degli archi ai nodi, visibili quando si seleziona un attributo o un gruppo di attributi. In altri modelli i magneti permetterebbero di connettere archi a gruppi di nodi, ma la definizione di schema graph impedisce questo tipo di relazione. I magneti sono i primi figli dei nodi ai quali sono attaccati nella gerarchia del modello del grafo. Per questo motivo si può utilizzare il metodo derivato dal `DefaultGraphCell getChildAt(0)`. Nelle funzioni del prototipo che fanno una scansione del grafo per visualizzarlo, quindi, in fase di implementazione si è utilizzato il comando Java `instanceof`, per verificare la classe specifica degli oggetti selezionati. Nell'implementazione del prototipo, quindi, è stato fatto largo uso del polimorfismo del Java, strumento potente che ci ha permesso di gestire tutti i nodi attraverso un'unica interfaccia comune, specificandone poi i diversi comportamenti a seconda delle necessità. Quando si seleziona un nodo del grafo, sia esso una proprietà o una misura, in realtà si seleziona anche il magnete, e durante l'eliminazione si deve provvedere alla cancellazione di entrambi.

E' importante capire la connessione gerarchica tra i nodi della rappresentazione grafica e gli attributi. Questa scelta è legata alla necessità di recuperare tutta la classe relativa ad un nodo, una volta che questo viene selezionato attraverso l'interfaccia grafica. Una `GraphCell`, infatti, non permette l'inserimento di dati aggiuntivi al nome, e se le due strutture dati non fossero state legate da una gerarchia di ereditarietà sarebbe stato necessario fare una ricerca su tutto il grafo per ogni operazione di modifica del livello dei meta-dati.

La classe `Attribute`, quindi, contiene anche le caratteristiche grafiche del nodo, come ad esempio `draw`, oltre a quelle specifiche per la modifica del nome (getName, setName) e del tipo di dato (setDataType, getDataType). `getPatentAttr` serve per trovare gli eventuali nodi che precedono nella gerarchia delle dipendenze l'attributo, mentre `getAttrTreeRoot` trova nella gerarchia l'attributo radice, che corrisponde alla dimensione nel caso delle proprietà e la misura nel caso delle misure derivate.

La classe `Measure` estende la classe `Attribute` con le funzioni relative alla gestione della formula di derivazione, che è obbligatoria solo nel caso delle misure derivate.

Il fact node concettualmente non è un vero e proprio attributo, ma rappresenta

l'evento centrale di tutto lo schema. Il comportamento di questo tipo di nodo è del tutto simile a quello degli attributi, tranne per il fatto che non può essere né eliminato né inserito. Per questo motivo in fase di progettazione è stato scelto di specificare con la classe del fatto la classe dell'attributo attraverso l'ereditarietà. Questo permette di evitare la creazione di un altro livello nella gerarchia dei nodi del grafo, che concettualmente potrebbe essere divisa in due differenti sotto-alberi:

1. { nodo }, { attributo }, { misura, proprietà }
2. { nodo }, { nodo del fatto }

Nel nostro caso, quindi, attributo e nodo sono utilizzati in maniera indistinta. Il concetto di nodo è stato eliminato completamente dalla modellazione in fase di progettazione per non creare confusione tra la classe `Node` del framework XML ed il concetto di attributo presente nel nostro prototipo.

Capitolo 6

Conclusioni

Niente basta a chi non basta ciò che è sufficiente.
(Epicuro)

Nell'ambito di questo lavoro di tesi abbiamo analizzato lo stato dell'arte dei database temporali relazionali ed il problema dell'evoluzione nel tempo delle strutture dei dati. Con l'evolversi del dominio applicativo e delle necessità di elaborazione dei dati acquisiti, si rendono necessari dei sistemi che riescano ad avvicinare la realtà al suo modello. Abbiamo visto come questa operazione porti al problema di garantire la possibilità di introdurre nuovi elementi nel modello ed al contempo mantenere i dati precedentemente inseriti. I vari sistemi analizzati si pongono ipoteticamente al centro di questi due requisiti contrastanti, avvicinandosi ad uno dei due estremi a seconda delle ipotesi prese in considerazione. Nel campo della business intelligence e dei data warehouse, la necessità di mantenere i dati anche in presenza di importanti modifiche alla loro struttura è primaria, ed una semplice gestione attraverso l'evoluzione spesso può portare a perdite consistenti di informazioni preziose.

Attraverso l'approccio introdotto in questo lavoro di tesi, si è dimostrato dal punto di vista formale la possibilità di gestire versioni diverse degli schemi che modellano il data warehouse. Successivamente sono state mostrate la progettazione dei meta-dati corrispondenti alle versioni degli schemi e la progettazione di un prototipo che dimostra la validità delle tesi introdotte.

Il prototipo è stato implementato con successo, come dimostrato dagli esempi di

funzionamento presenti in alcune immagini utilizzate nel precedente capitolo. Alcune delle scelte progettuali iniziali sono state modificate sulla base di alcuni requisiti introdotti dai due framework utilizzati (JGraph e Dom4J), ma nel complesso la modellazione iniziale non ha subito variazioni sostanziali.

L'enorme flessibilità di questo approccio sconta solamente il difetto di richiedere un grosso intervento da parte del progettista. Alcune operazioni, infatti, non possono essere totalmente automatizzate, come per esempio la propagazione delle modifiche negli schemi aumentati delle versioni precedenti. D'altra parte questo tipo di operazioni non è molto frequente, perché coinvolge solamente le modifiche a livello strutturale del data warehouse: sta quindi al progettista capire quando è necessario introdurre una nuova versione ed adattare il sistema alle mutate esigenze. Anche in un ambiente molto dinamico, questo per fortuna non accade quotidianamente, quindi la maggior parte dell'utilizzo del sistema introdotto in questo lavoro di tesi riguarderà l'interrogazione delle diverse versioni.

La gestione delle versioni attraverso gli schemi aumentati e gli schemi comuni rende totalmente trasparente il sistema e le elaborazioni necessarie all'utente finale, e quindi soddisfa pienamente il requisito iniziale della semplicità d'uso per l'analista che utilizza il nostro sistema.

I possibili sviluppi futuri del prototipo progettato ed implementato attraverso questo lavoro di tesi potranno riguardare la gestione delle sessioni ROLAP per l'interrogazione delle sorgenti di dati e la gestione delle operazioni di migrazione dei dati estensionali corrispondenti alle diverse versioni degli schemi. Le operazioni di migrazione, in particolare, non potranno essere totalmente automatizzate, ma si dovrà prevedere un supporto al progettista che permetta di determinare le azioni da effettuare, possibilmente nascondendone i dettagli implementativi ed i controlli sulla consistenza dei risultati.

Bibliografia

[AGU72] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

[BEK+04] B. Bębel, J. Eder, C. Koncilia, T. Morzy, and R. Wrembel. Creation and management of versions in multiversion data warehouse. In *Proc. SAC*, pages 717–723, Nicosia, Cyprus, 2004.

[CGS97] Cristina De Castro, Fabio Grandi, and Maria Rita Scalas. Schema versioning for multitemporal relational databases. *Inf. Syst.*, 22(5):249–290, 1997.

[DGS95] C. De Castro, F. Grandi, and M. R. Scalas. On schema versioning in temporal databases. In S. Clifford and A. Tuzhilin, editors, *Recent advances in temporal databases*, pages 272–294. Zurich, Switzerland, 1995.

[EK01] J. Eder and C. Koncilia. Changes of dimension data in temporal data warehouses. In *Proc. DaWaK*, pages 284–293, 2001.

[EKM02] J. Eder, C. Koncilia, and T. Morzy. The COMET metamodel for temporal data warehouses. In *Proc. CAiSE*, pages 83–99, 2002.

[GMR98] M. Golfarelli, D. Maio, and S. Rizzi. The Dimensional Fact Model: a conceptual model for data warehouses. *Int. Journal of Cooperative Information Systems*, 7(2-3):215–247, 1998.

[Gra02] F. Grandi. A relational multi-schema data model and query language for full support of schema versioning. In *Proc. 10th SEBD*, Isola d'Elba, Italy, 2002.

[Gro96] Stanford Technology Group. Designing the data warehouse on relational databases. 1996.

[J+98] C. Jensen et al. The consensus glossary of temporal database concepts. In O. Etzion, S. Jajodia, and S. Sripada, editors, *Temporal Databases: Research and Practice*, pages 367–405. Springer, 1998.

[JCE+94] C. Jensen, J. Clifford, R. Elmasri, S. K. Gadia, P. J. Hayes, and S. Jajodia. A consensus glossary of temporal database concepts. *ACM SIGMOD Record*, 23(1):52–64, 1994.

[JJQV99] Matthias Jarke, Manfred A. Jeusfeld, Christoph Quix, and Panos Vassiliadis. Architecture and quality in data warehouses: An extended repository approach. *Information Systems*, 24(3):229–253, 1999.

[Lec04] J. Lechtenbörger. Computing unique canonical covers for simple fds via transitive reduction. *Information Processing Letters*, 92(4):169–174, 2004.

[LH90] Barbara Staudt Lerner and A. Nico Habermann. Beyond schema evolution to database reorganization. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 67–76, New York, NY, USA, 1990. ACM Press.

[LHV02] C. Letz, E. Henn, and G. Vossen. Consistency in data warehouse dimensions. In *Proc. IDEAS*, pages 224–232, 2002.

[Lie83] Bennet P. Lientz. Issues in software maintenance. *ACM Comput. Surv.*, 15(3):271–278, 1983.

[LV03] J. Lechtenbörger and G. Vossen. Multidimensional normal forms for data warehouse design. *Information Systems*, 28(5):415–434, 2003.

[Mai83] D. Maier. *The theory of relational databases*. Computer Science Press, 1983.

[MS90] E. McKenzie and R. Snodgrass. Schema evolution and the relational algebra. *Information Systems*, 15(2):207–232, 1990.

[PS03] E. Pourabbas and A. Shoshani. Answering joint queries from multiple aggregate OLAP databases. In *Proc. 5th DaWaK*, Prague, 2003.

[Qui99] Christoph Quix. Repository support for data warehouse evolution. In *Design and Management of Data Warehouses*, page 4, 1999.

[RCR93] John F. Roddick, Noel G. Craske, and Thomas J. Richards. A taxonomy for schema versioning based on the relational and entity relationship models. In *International Conference on Conceptual Modeling / the Entity Relationship Approach*, pages 137–148, 1993.

[Rod95] John F. Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383–393, 1995.

[RS95] John F. Roddick and Richard T. Snodgrass. Schema versioning. In *The TSQL2 Temporal Query Language*, pages 425–446. 1995.

[ST82] Ben Shneiderman and Glenn Thomas. An architecture for automatic relational database sytem conversion. *ACM Trans. Database Syst.*, 7(2):235–257, 1982.

[TSQ95] TSQL2 Language Design Committee. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 1995.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge - Base Systems - Volume 1: Classical Database Systems*. Computer Science Press, 1988. ULL j 88:1 P-Ex.

[VMRC02] A. Vaisman, A. Mendelzon, W. Ruaro, and S. Cymerman. Supporting dimension updates in an OLAP server. In *Proc. CAiSE*, pages 67–82, 2002.

[VS99] Panos Vassiliadis and Timos K. Sellis. A survey of logical models for OLAP databases. *SIGMOD Record*, 28(4):64–69, 1999.

[W3C01] W3C. Xml schema recommendation. 2001.

[Yan01] J. Yang. *Temporal data warehousing*. PhD thesis, Stanford University, 2001.

Indice analitico