UNIVERSITÀ DEGLI STUDI DI PISA

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

Corso di Laurea Specialistica in Tecnologie Informatiche

TESI DI LAUREA

Analisi e sperimentazione di curve space-filling per range query multiattributo in sistemi P2P

CANDIDATO:
Matteo Parchi

RELATORI:
Prof.ssa Laura Ricci
Prof. Massimo Coppola

CONTRORELATORE:
Prof. Roberto Grossi

ANNO ACCADEMICO 2009/2010

Indice

Capitolo 1

Introduzione

Le applicazioni distribuite basate sul modello P2P (*Peer-to-Peer*) [1] sono state negli ultimi anni oggetto di una grande diffusione nel settore informatico. Tali sistemi sono caratterizzati dal fatto che tutti i nodi sono nodi equivalenti, capaci di auto-organizzarsi e in grado di condividere un insieme di risorse distribuite all'interno della rete. Tali nodi sono chiamati *peer* ad evidenziarne il grado di parità dei ruoli. L'evoluzione delle applicazioni ha visto l'adozione, da parte degli sviluppatori, di stili architetturali che si distaccassero sempre più dal modello centralizzato *client/server*. Il paradigma computazionale delle reti P2P si basa sulla *decentralizzazione* del controllo e sulla condivisione delle risorse. L'ambiente di esecuzione delle reti P2P è estremamente *dinamico* e può assumere grandi dimensioni su larga scala. L'aspetto maggiormente critico delle reti client/server è la presenza di un punto di fallimento e la limitata scalabilità delle applicazioni su reti di grande scala. Le reti P2P non pretendono di sostituirsi a queste reti, ma piuttosto di fornire una valida alternativa in tutti quei contesti in cui le criticità del modello consolidato client/server rappresentano un problema.

La prima generazione di applicazioni P2P come Napster [3] e Gnutella [2] ha contribuito alla diffusione del paradigma P2P e ne ha evidenziato le enormi potenzialità, in particolare le capacità di aggregazione di una grande mole di dati, ma anche la possibilità di condividere potenze di calcolo in maniera poco costosa. Con l'evoluzione delle potenzialità della rete, a causa della *topologia non strutturata* della rete, le applicazioni P2P di prima generazione hanno mostrato i primi limiti. Un sistema P2P non strutturato è caratterizzato dal fatto che ogni risorsa è localizzata sul peer che la mette a disposizione e non esistono informazioni relative alla loro locazione. A causa

della mancanza di un'organizzazione all'interno del sistema, la ricerca risulta essere difficoltosa. D'altra parte, l'inserimento o la rimozione di nodi dalla rete risulta essere un'operazione semplice e poco costosa.

La seconda generazione di applicazioni P2P è basata su *reti di tipo strutturato*. Le reti P2P strutturate utilizzano per la costruzione della topologia di rete tra i propri peer una struttura di base con specifiche proprietà, in modo tale da ottimizzare le operazioni di ricerca. La principale struttura di base utilizzata per la costruzione della topologia delle reti P2P di seconda generazione è la *Distributed Hash Table, DHT* [8] [9] [10] [11]. Le DHT, attraverso l'impiego di una funzione hash, distribuiscono i peer e le risorse sulla rete in maniera casuale. Grazie all'utilizzo di tali strutture, si è arrivati ad avere *costi di ricerca logaritmici* organizzando la rete in base ai contenuti attraverso la costruzione di un indice distribuito. Per contro, l'*espressività delle query è limitata* a ricerche di tipo esatto e su singolo attributo. Avendo un indice distribuito unidimensionale non risulta praticabile eseguire, ad esempio, range query o ricerche su un numero arbitrario di attributi senza ottenere costi di ricerca esponenziali. Negli ultimi anni si è assistito all'utilizzo di applicazioni P2P in nuovi contesti quali il *Gaming online* [12] o il *Grid Computing* [13]. Queste tipologie di applicazioni presentano un insieme di caratteristiche comuni quali ad esempio l'elevata dinamicità delle informazioni, la rappresentazione di una risorsa mediante più attributi o la ricerca di risorse mediante criteri di selezione strutturati. Se consideriamo le applicazioni P2P in un contesto come il *Grid Information Service* [7], appare chiaro come tali applicazioni debbano effettuare un processo di discovery delle risorse condivise in maniera immediata, efficiente, scalabile, ma soprattutto in maniera espressiva. L'espressività di ricerca risulta il punto cruciale delle applicazioni P2P. Nel caso del Grid Information Service, contesto in cui la presente tesi si è focalizzata, risulta fondamentale fornire un supporto per l'esecuzione di query del tipo: *Find N resource with cpuSpeed* $\geqslant$ *2.0Ghz and 0.4* $\leqslant$ *load* $\leqslant$ *0.8 and freeRam* $\geqslant$ *20%*. Gli attuali modelli P2P non sono in grado di soddisfare tali requisiti e anzi risultano fortemente limitati, per quanto riguarda l'espressività, nel processo di ricerca delle risorse.

In letteratura sono state presentate diverse proposte, [15] [16] [17] [18] [19], che si occupano di potenziare i modelli P2P incrementandone l'espressività di ricerca delle risorse. Tali proposte sono classificabili all'interno di due aree di ricerca:

1. proposte che modificano le DHT esistenti,

2. proposte che sostituiscono la struttura di base DHT attraverso strutture distribuite di tipo gerarchico.

I lavori basati sulla modifica delle DHT esistenti si focalizzano nel tentativo di introdurre all'interno della DHT il concetto di *località dei dati* [16]. La rete P2P è costruita attraverso l'utilizzo della DHT e il supporto a ricerche più espressive, quali ad esempio quelle effettuate mediante range query, viene fornito attraverso la manipolazione della funzione hash della DHT. La funzione hash dovrà preservare la località dei dati. Nel lavoro presentato da [15], *MAAN Multi-Attribute Addressable Network*, è evidente come il centro del modello prestazionale in tali sistemi sia la scelta della funzione hash. In presenza di distribuzioni non uniformi tali modelli iniziano a mostrare problematiche di *load-balancing*.

Sulla stessa tipologia di approccio si trovano i lavori basati sull'utilizzo di curve space-filling [22] [29] [32] per effettuare la linearizzazione di uno spazio multi-attributo in uno spazio uni-dimensionale, preservando al meglio la proprieta di località dei dati originari. Mediante tale approccio è possibile definire un supporto che consenta di poter effettuare range query multiattributo sulla DHT. In questa direzione si colloca il lavoro di ricerca proposto da [16] con il sistema *Squid*. Squid è una rete P2P strutturata che utilizza una DHT e memorizza le risorse come chiavi derivate generate dall'applicazione della funzione space-filling di Hilbert. Tale approccio delega il compito di distribuzione delle risorse nella rete alla funzione space filling, ma memorizza le chiavi derivate direttamente sull'anello Chord [8] e ciò porta ad una estrema sensibilità in caso di distribuzioni non uniformi dei dati. A causa delle problematiche appena esposte, la ricerca si è indirizzata verso un'altra direzione, basandosi sull'utilizzo di strutture di base alternative alle DHT, strutture di tipo gerarchico e distribuite. Ad esempio, in *BATON* [17], la struttura utilizzata è basata su un *B-albero* e tenta di sfruttare le proprietà possedute dalle strutture gerarchiche, già note ed utilizzate in ambienti non distribuiti, per incrementare l'espressività delle operazioni di ricerca nei modelli P2P. L'approccio proposto in BATON presenta degli evidenti overhead di gestione soprattutto in presenza di attributi dinamici per effetto del continuo bilanciamento della struttura distribuita. Risulta pertanto fondamentale che le proposte siano in grado di gestire efficacemente il dinamismo tipico dei sistemi P2P senza introdurre elevati overhead di gestione e colli di bottiglia verso particolari peer della rete. Un approccio che si spinge in questa direzione è *Cone* [19]. Cone rappresenta una soluzione ibrida, infatti, la strutturazione della rete avviene attraverso una DHT, ma al

contempo mantiene una struttura dati distribuita basata su un albero distribuito ed a differenza delle altre soluzioni risulta essere molto promettente sia per quanto riguarda il bilanciamento del carico che per la gestione della struttura in caso di aggiornamenti. Per contro, in Cone è possibile effettuare solamente una ristretta tipologia di query del tipo *Trova k risorse di dimensione maggiore di S* e ciò riduce parecchio l'espressività di ricerca. Una soluzione ispirata a Cone, in grado di gestire il dinamismo delle risorse e che permetta di effettuare ricerche più espressive attraverso range query è *XCone*. Proposto in [44], XCone integra la DHT con una struttura dati gerarchica il cui scopo è definire un albero logico di aggregazione delle chiavi distribuite sulla DHT che guidi la ricerca delle soluzioni di una range query unidimensionale. XCone eredita dalla DHT le proprietà di bilanciamento del carico tra i peer della rete in quanto non modifica la struttura della DHT stessa, ma utilizza un *trie* [21] costruito sugli identificativi della DHT, ma mantenuto in uno spazio distinto come struttura dati gerarchica aggiuntiva. Quando un peer si unisce alla rete XCone ottiene un identificatore dalla DHT e di conseguenza viene associato al rispettivo nodo foglia del trie per tutta la permanenza del peer in rete. In XCone le risorse sono memorizzate localmente nei peer che rappresentano le foglie dell'albero e ogni nodo logico interno dell'albero logico viene assegnato ad un peer attraverso una funzione di mapping. I descrittori delle risorse vengono aggregati secondo una funzione di aggregazione. La funzione di mapping di XCone determina quindi quali nodi logici non foglia vengono assegnati ad ogni peer della rete. La funzione di digest permette di associare, ad ogni nodo interno dell'albero, una struttura che sintetizza le chiavi contenute nel sottoalbero radicato nel nodo interno. XCone permette di eseguire range query su singolo attributo sulla DHT.

La presente tesi, dopo aver analizzato gli approcci proposti in letteratura, si è focalizzata sulla definizione di un supporto per range query multiattributo da integrare in XCone. Il risultato della tesi è la definizione del sistema *HASP, Hierachical Aggregation over Space-Filling*. *HASP* è basato sull'uso di curve space-filling per la generazione di una chiave derivata a partire dal valore di n attributi che definiscono la singola risorsa. Dopo l'analisi di numerose curve space-filling e delle loro *proprietà di clustering* [35], riportata nel capitolo 3, è stata scelta la *curva Z-order* [30]. Anche se la *Z-order curve* preserva un minor grado di località rispetto alla curva di Hilbert, essa rappresenta un buon compromesso perchè caratterizzata da maggiore semplicità di definizione degli algoritmi di generazione della chiave derivata e di risoluzione di range query. La complessità dell'algoritmo di generazione della chiave derivata è $O(n\,k)$,

che è la complessità intrinseca indotta dalle caratteristiche del problema. Le curve space-filling attraversano tutto lo spazio n-dimensionale delle risorse e permettono di identificare in modo univoco ciascuna n-upla di coordinate linearizzando uno spazio n-dimensionale e associando un valore univoco a ciascun punto di tale spazio. Tale valore prende il nome di *chiave surrogata* o *chiave derivata*. La località dei dati è preservata con gradi diversi a seconda della curva space-filling specifica, ma è caratteristica di queste curve che *chiavi derivate vicine sulla curva siano generate a partire da punti più o meno vicini nello spazio n-dimensionale*. La generazione della chiave surrogata, come vedremo nel capitolo 4 per la curva di Hilbert [22] e per la curva Z-order [24], è differente in base al tipo di curva space-filling e può richiedere l'esecuzione di algoritmi complessi. L'elevata modularità nell'implementazione di *HASP* rende possibile modificare la curva space-filling da utilizzare a seconda delle esigenze. Lo spazio dei valori delle chiavi derivate risulta di dimensione notevole. Valori tipici per la classe di applicazioni considerata producono uno spazio di chiavi derivate la cui dimensione può essere pari a 2^{60}. La dimensione di questo spazio implica la definizione di nuove funzioni di digest differenti rispetto a quelle proposte originariamente da XCone. Per poter fornire informazioni significative riguardo ad uno spazio delle chiavi di dimensione così grande come può essere quello generato da una funzione space-filling, è necessaria la definizione di una *struttura dati dinamica* per la memorizzazione del digest. L'informazione di digest contenuta nei nodi dell'albero di aggregazione *HASP* viene utilizzata per calcolarne l'intersezione con la range query, rappresentata geometricamente da un iper-rettangolo. Per la risoluzione delle range query saranno presentati nel capitolo 6 due approcci. Un primo approccio calcola tutti i segmenti della curva space-filling contenuti nell'iper-rettangolo definito dalla range query e li utilizza per risolvere la range query multiattributo. Un secondo approccio sfrutta la struttura gerarchica costruita sopra la DHT ed inoltra la range query solamente a quei nodi dell'albero la cui informazione di digest assicuri la presenza nel sotto-albero radicato in tali nodi di almeno una risorsa che la soddisfi. Questo secondo approccio, *guidato dai dati*, sfrutta le modifiche alle funzioni di digest descritte nel capitolo 5. Nel capitolo 6 è proposto un algoritmo per il calcolo dell'intersezione tra l'iper-rettangolo definito da una range query e un iper-quadrante corrispondente ad un segmento di curva Z-order. Tale algoritmo sfrutta le proprietà di identificazione dei quadranti di tale curva space-filling. Integrati in *HASP*, gli algoritmi presentati ai capitoli 4, 5 e 6, permettono la risoluzione di range query multiattributo e rimuovono così il vincolo dell'unidimensionalità

del sistema XCone, pur mantenendone inalterate le caratteristiche peculiari.

L'organizzazione della tesi è la seguente. Nel capitolo 2 sono presentati i principali approcci presenti in letteratura per la gestione di query complesse in reti P2P. Le varie proposte verranno analizzate e ne saranno evidenziati vantaggi e limiti. Nel capitolo 3 verranno presentate le curve space-filling e descritte in particolare la curva di Hilbert, la curva Z-order e la curva Gray-code, effettuando un confronto tra le loro proprietà di clustering. Nel capitolo 4 verrano presentati e confrontati gli algoritmi di generazione della chiave derivata utilizzando le curve space-filling di Hilbert e Z-order. Nel capitolo 5 verrà presentata l'architettura di *HASP*. Saranno illustrate le funzioni di digest realizzate per l'impiego con chiavi derivate di grande dimensione. Nel capitolo 6 verrà descritto il processo di risoluzione di una range query multidimensionale in *HASP*, mentre nel capitolo 7 verrà illustrata l'implementazione del sistema e lo pseudo-codice degli algoritmi proposti. Nel capitolo 8 saranno descritti gli esperimenti per la valutazione di *HASP* e i risultati sperimentali ottenuti. Infine nel capitolo 9 saranno riassunte alcune considerazioni conclusive e i possibili sviluppi futuri del sistema.
I contributi originali di questa tesi sono i seguenti:

1. *sviluppo* degli algoritmi per la generazione della chiave derivata sia per la curva di Hilbert che per la curva Z-order;

2. definizione di diverse *strategie* per la risoluzione di range query multiattributo mediante l'intersezione tra l'iper-rettangolo che definisce la query e la curva space-filling;

3. *integrazione* degli algoritmi definiti in 1) e 2) in XCone. Il sistema risultante, *HASP*, è stato valutato con dati reali ricavati dalla piattaforma distribuita *PlanetLab* [43].

Capitolo 2

Supporto di query complesse in sistemi P2P: stato dell'arte

In questo capitolo verranno illustrati gli attuali approcci presenti in letteratura. In particolare verranno classificate le diverse proposte, evidenziandone i vantaggi ed i limiti.

2.1 Introduzione

Le reti P2P rappresentano attualmente una promettente soluzione per la condivisione di risorse in ambienti distribuiti. Le Distributed Hash Table (DHT) sono alla base di numerosi servizi quali il file sharing, lo storage network ed il content delivery network. Negli ultimi anni si è inoltre assistito all'applicazione di tali sistemi verso nuovi contesti come il Gaming-online [12], il Semantic Web o il Grid Computing [7]. Una ben nota problematica delle reti P2P verso la quale la ricerca scientifica si è focalizzata riguarda la mancanza di espressività nei criteri di selezione delle risorse. Il recupero efficiente di un insieme di risorse che soddisfino dei criteri prefissati si rivela un requisito fondamentale sia nei sistemi di tipo Grid che nelle applicazioni di gaming online. Consideriamo il caso dei sistemi Grid. Un sistema Grid fornisce un'infrastruttura di base indispensabile per la condivisione di un insieme di risorse, quali desktops, clusters computazionali, supercomputers, sensori o strumenti scientifici. Una delle funzionalità di base, fornite dall'infrastruttura, deve permettere di selezionare su larga scala un insieme di risorse in accordo con specifici criteri impostati dall'utente (*Resource Di-*

scovery/Selection). In questi nuovi ambiti si sta affermando sempre di più l'utilizzo di reti P2P che sollevano nuovi interrogativi e problematiche da risolvere per la comunità scientifica.

2.2 Query complesse in sistemi P2P: problemi aperti

Tramite l'utilizzo delle DHT, le reti Peer-to-Peer sono in grado di fornire un'infrastruttura scalabile e con funzionalità di base per la ricerca delle risorse. Le query, supportate da questi sistemi, sono del tipo: *Trova tutti i file il cui nome contenga una determinata stringa*. Per poter estendere l'utilizzo delle reti P2P verso nuovi contesti è necessario aumentare l'espressività del linguaggio di interrogazione in modo tale da riuscire ad esprimere correlazioni e/o combinazioni tra le proprietà delle risorse.

Una prima classificazione delle query è basata sul numero di attributi in esse specificati e conduce alla distinzione tra:

1. Query monodimensionali o a singola dimensione, nel caso in cui il criterio di ricerca coinvolga un solo attributo.

2. Query multidimensionali, nel caso in cui il criterio di ricerca sia dato dall'unione dei vincoli su tutti o alcuni degli attributi che compongono le risorse.

E' facilmente intuibile come le query multidimensionali risultino significativamente più complesse e diano origine a numerose problematiche. Operando un'ulteriore classificazione, in base al criterio di selezione delle risorse sulla rete, possiamo suddividere l'insieme delle query multidimensionali in:

- **Exact match query**, in cui vengono specificati i valori esatti da ricercare per tutti gli attributi che compongono la risorsa. Ad esempio: *Trova delle risorse che abbiano* $OS = Linux$, $CPU = 2Ghz$, $RAM = 1GB$, $MemLibera = 512MB$ *e* $LOAD = 30\%$*(Carico di lavoro).*

- **Partial match query**, come le exact match query, ma in cui vengono specificati i valori esatti da ricercare solo per un sotto-insieme degli attributi che compongono la risorsa. Ad esempio, supponendo che una risorsa sia definita dai seguenti cinque attributi: *OS, CPU, RAM, memoria disponibile e carico di lavoro corrente (LOAD)*, la query risulterà del tipo *Trova delle risorse che abbiano* $OS = Linux$ *e* $CPU = 2Ghz$ *e* $RAM = 1GB$.

- **Range query**, in cui viene specificato per tutti gli attributi o solo per un sotto-insieme di essi, un range di valori ammissibili. Ad esempio: *Trova delle risorse che abbiano* $2Ghz \leqslant CPU \leqslant 3Ghz$ *e* $2GB \leqslant RAM \leqslant 4GB$ *e* $0\% \leqslant LOAD \leqslant 30\%$.

- **Boolean query**, in cui si specificano le condizioni su tutti o alcuni degli attributi attraverso l'utilizzo di operatori booleani. Ad esempio: *Trova delle risorse che (not* $RAM \leqslant 1GB$*) and (not* $OS = Linux$*).*

- **Nearest-Neighbor Query**, in cui vengono specificati per un insieme di attributi sia un insieme di condizioni che una funzione di valutazione delle risorse trovate definita *metrica*. La metrica consente di caratterizzare le risorse rispetto alla distanza dalla soluzione ottimale di ricerca. Ad esempio: *Trova delle risorse che abbiano* $CPU \geqslant 2Ghz$ *e* $1GB \leqslant RAM \leqslant 2GB$ *considerando come metrica la distanza dello spazio euclideo d-dimensionale.*

La classificazione delle query come sopra descritta è mostrata in figura 2.1.

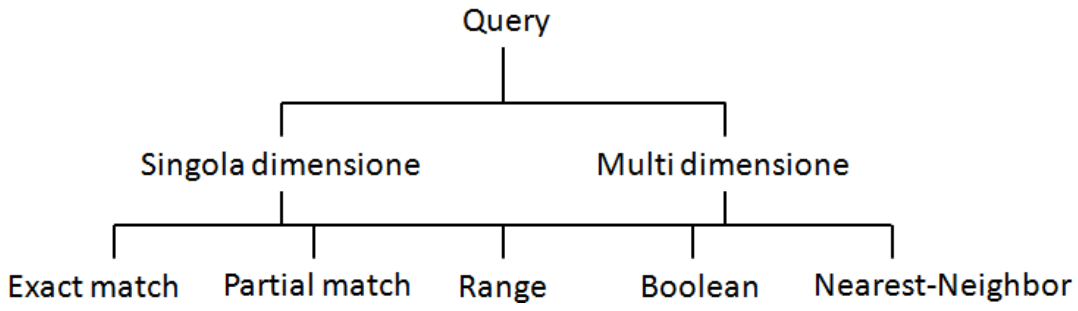


Figura 2.1 – Classificazione query

Nella ricerca delle risorse che soddisfano dei determinati vincoli sul valore degli attributi che le compongono, è necessario tener conto anche della *dinamicità* delle risorse. Riprendendo l'esempio del Resource Discovery utilizzato in precedenza, una query del tipo *Trova k risorse con OS=Linux e 0%* $\leqslant$ *LOAD*$\leqslant$ *30% e CPU* $\geqslant$ *2Ghz e 1GB* $\leqslant$ MemoriaLibera$\leqslant$ *4GB,* è composta da attributi, come il carico di lavoro corrente sulla cpu e la quantità di memoria fisica disponibile, i cui valori sono fortemente variabili nel tempo. Si può dunque operare un'ulteriore classificazione su ciascuno degli attributi che caratterizzano una risorsa:

1. **Attributi statici**, la cui frequenza di aggiornamento rimane tendenzialmente costante nel tempo. Appartengono a questo tipo di categoria, attributi quali il sistema operativo o la frequenza di clock della cpu.

2. **Attributi dinamici**, i cui valori possono variare significativamente istante per istante. Esempi di questo tipo di attributi sono il carico di lavoro corrente sulla cpu, la memoria fisica disponibile o la percentuale di utilizzo della rete.

Gli obiettivi su cui si sta concentrando la ricerca riguardano il potenziamento delle DHT al fine di migliorare ed aumentare l'espressività del linguaggio di query delle risorse sulla rete e la gestione dei vari tipi di attributi.

2.3 Classificazione sistemi P2P

L'architettura di una rete di tipo P2P risulta molto differente rispetto ad una rete classica di tipo client/server e cerca di essere quanto più possibile *scalabile, tollerante ai guasti e completamente decentralizzata*, sfruttando al massimo le potenzialità della rete di interconnessione Internet.

Possiamo analizzare le reti P2P secondo due aspetti:

1. La topologia delle rete

2. L'organizzazione delle strutture dati

2.3.1 Topologia della rete

La topologia della rete indica la struttura mediante la quale i peer sono logicamente interconnessi e può essere di due tipi: *strutturata* e *non strutturata.*

Reti non strutturate. Una rete P2P non strutturata è solitamente descritta da un grafo in cui le connessioni dei peer sono basate sulla loro popolarità. Tra le varie reti P2P non strutturate, quali Gnutella [2], Napster [3], BitTorrent [4], JXTA [5] e Kazaa [6] possiamo distinguere vari livelli di decentralizzazione, dove, per livello di decentralizzazione, si identifica la possibilità di effettuare in maniera distribuita una ricerca e/o recupero delle risorse.

Una classificazione delle reti non strutturate può essere data dal tipo di ricerca che esse utilizzano: *deterministica* o *non deterministica*. Nel caso di ricerca deterministica, ciascuna risorsa, individuata da una query, viene localizzata in tempi certi; fanno parte di questa categoria di sistemi P2P la rete Napster e quella BitTorrent. Nel caso di ricerca non deterministica, reti P2P, quali Kazaa e Gnutella, mantengono un sistema di indici distribuito ed effettuano la ricerca delle risorse tramite un modello basato sul *flooding* dei messaggi ai vicini sulla rete. Ad esempio, in Kazaa, esistono due tipi di peer: i peers normali e i super-peers. I super-peers raccolgono i dati ricevuti dai peer normali, eseguono un algoritmo di compressione delle informazioni ed eseguono il flooding dei messaggio di ricerca esclusivamente verso altri super-peers. Tali reti limitano il numero di messaggi scambiati sulla rete, ma risultano poco scalabili e presentano la problematica dei *falsi positivi* che riduce l'accuratezza delle operazioni di ricerca.

Reti strutturate. Appartengono alla categoria delle reti P2P strutturate, le reti CHORD [8], CAN [9], Pastry [10], Tapestry [11] e Kademlia [14] le quali utilizzano una struttura di base per la costruzione della rete. Nella maggior parte dei casi, la struttura utilizzata è la DHT o Distributed Hash Table, che garantisce un modello di ricerca di tipo deterministico e una complessità quasi sempre logaritmica nel numero di nodi presenti nella rete per le operazioni di ricerca e/o di inserzione. Le differenze che vi sono tra le reti P2P strutturate sopra citate, che utilizzano le DHT, sono da ricercarsi nei diversi algoritmi utilizzati per il posizionamento dei nodi nella rete, nell'assegnazione delle risorse ai nodi e nella ricerca di queste sulla rete DHT.

Uno schema riassuntivo delle reti P2P in cui viene evidenziata la distinzione tra reti strutturate e non strutturate, è visibile in figura 2.2.

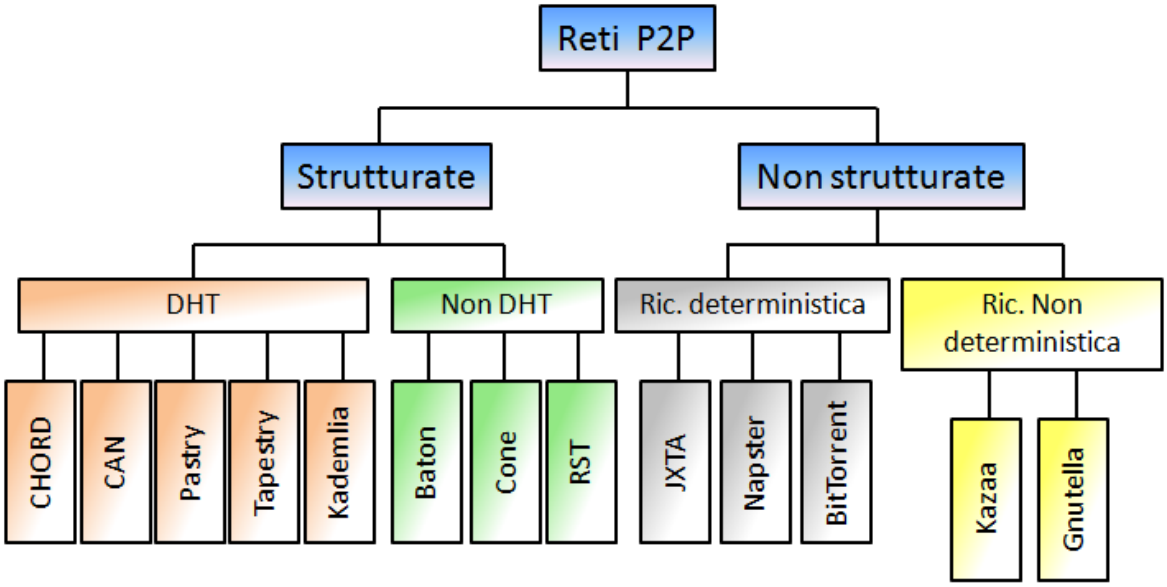


Figura 2.2 – Topologia reti P2P

2.3.2 Organizzazione delle strutture dati

Le DHT costituiscono la struttura dati più utilizzata nelle reti P2P strutturate, in quanto garantiscono anche un supporto perfetto per l'esecuzione di query di tipo - *Exact Match* (vedi sezione 2.2). L'estensione degli attuali sistemi esistenti comporta necessariamente una ristrutturazione o potenziamento delle strutture dati attuali e, per fare ciò, in letteratura sono state individuate tre principali direzioni di ricerca che possono essere seguite:

1. utilizzo di tecniche di hashing con relative varianti
2. utilizzo di curve di tipo space-filling
3. utilizzo di strutture dati basate su alberi di ricerca

Tecniche di hashing. Il primo approccio, basato sulla manipolazione delle funzioni hash, consiste nel cercare di *preservare alcune proprietà importanti come l'ordine o la località dei dati* in modo che la risoluzione delle query possa trarne vantaggio, sfruttando differenti euristiche studiate ad hoc. Tale approccio può dar luogo a criticità che devono essere opportunamente gestite in particolare nel caso di distribuzione non uniforme dei dati.

Curve space-filling. Il secondo approccio, basato sull'utilizzo di strutture dati, quali le *curve space-filling*, permette di effettuare un *mapping* da uno spazio n-dimensionale verso uno spazio uni-dimensionale o lineare. Una risorsa, definita da un insieme di n attributi, viene posizionata in uno spazio lineare tramite una specifica funzione di mapping differente a seconda della curva space-filling utilizzata. Il valore generato da tale funzione viene chiamato *chiave derivata* o *chiave surrogata*. Per una trattazione più approfondita delle proprietà delle varie curve space-filling si rimanda il lettore al capitolo 3.

Alberi di ricerca distribuiti. Il terzo approccio è basato sull'utilizzo di strutture dati quali gli *alberi di ricerca distribuiti,* che possono essere suddivisi in due categorie distinte a seconda che si operi con dati lineari o dati multidimensionali. Nel caso di dati ad una dimensione, le strutture dati distribuite tendono a basarsi sull'utilizzo di *Hash Tree* o *B-alberi*, mentre nel caso di dati in uno spazio n-dimensionale si hanno strutture dati ispirate ad alberi *Space Driven* come i *KD-Tree* o a strutture *Data Driven* quali gli *R-Tree*. In tutti questi casi le operazioni di ricerca non riportano falsi negativi.

Le proprietà auspicabili, che tutte le strutture dati dovrebbero possedere, sono due: *il bilanciamento del carico* in modo che ciascun nodo della rete in media mantenga una quantità di dati paragonabile e scambi un numero simile di messaggi con gli altri nodi e la *minimizzazione delle informazioni di stato,* in modo che ciascun nodo debba mantenere una quantità minima di informazioni per la gestione della struttura dati distribuita.

In figura 2.3 è mostrato uno schema dell'organizzazione delle strutture dati nei sistemi P2P.

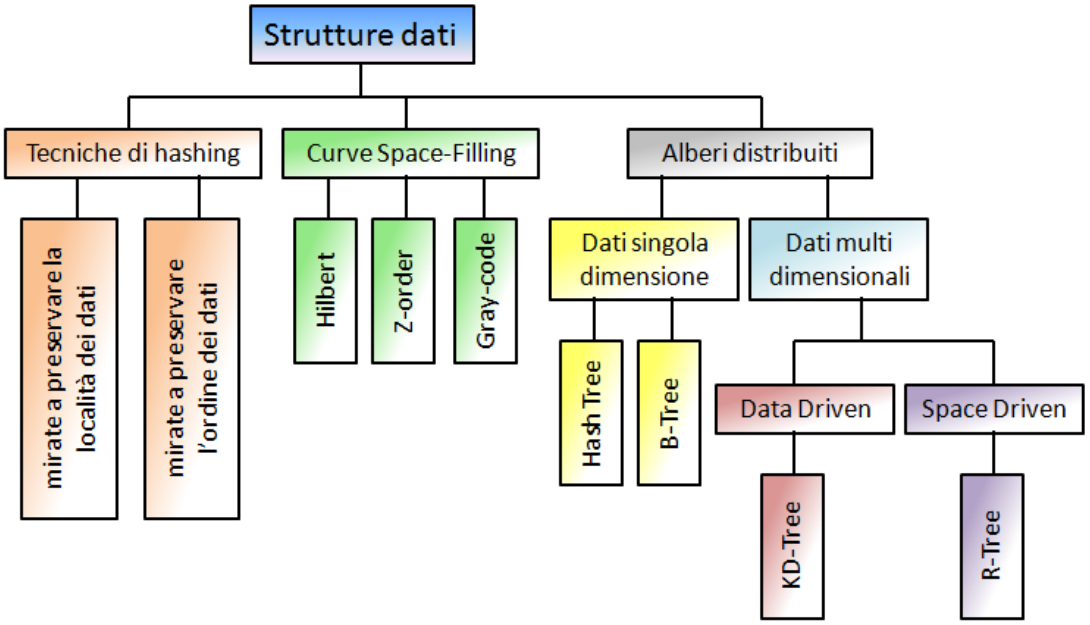


Figura 2.3 – Organizzazione delle strutture dati P2P

2.4 Proposte in letteratura

In questa sezione sono illustrati gli approcci principali per ognuna delle direzioni di ricerca presentate nei paragrafi precedenti.

2.4.1 MAAN

MAAN [15] o Multi-Attribute Addressable Network è una rete P2P strutturata in grado di fornire supporto a range query multidimensionali. MAAN utilizza un anello Chord ed estende questa DHT tramite l'impiego di una funzione di hash in grado di preservare la località dei dati. Le risorse sono identificate da un insieme di coppie attributo-valore

$$(a_1, v_1), (a_2, v_2), ..., (a_n, v_n)$$

memorizzate secondo la funzione di hashing sull'anello Chord $[0; 2^{m-1}]$. Il processo di memorizzazione sulla rete consiste quindi nell'applicare per ogni coppia la relativa funzione hash e nel memorizzare, nel corrispondente nodo della rete, l'intera risorsa. Da notare come una risorsa venga registrata un numero di volte pari al numero di

attributi che la compongono. L'esecuzione di una range query multidimensionale in MAAN comporta la risoluzione di un insieme di query unidimensionali, una per ogni coppia attributo-valore della risorsa. Una volta ottenuti i risultati, la risoluzione della range query multidimensionale consiste in una semplice intersezione. Al fine di ottimizzare il processo appena descritto, gli autori propongono il concetto di *attributo dominante*. Un attributo è definito dominante se la percentuale di nodi filtrati risulta maggiore rispetto a quella degli altri attributi. Il nuovo processo di risoluzione di un'interrogazione consisterà nel risolvere esclusivamente la query unidimensionale sull'attributo dominante. Ricordando che ogni nodo memorizza l'intera risorsa, la ricerca per attributo dominante potrà risolvere localmente l'intera query multidimensionale, evitando l'invio di ulteriori messaggi.

2.4.2 Squid

Squid [16] è una rete P2P che utilizza la curva space-filling di Hilbert per memorizzare le risorse all'interno di un anello Chord e preservare la località dei dati. Squid permette dunque di effettuare range query multidimensionali. Per stabilire l'allocazione di una risorsa sull'anello Chord, si calcola la sua chiave derivata tramite l'algoritmo di mapping della curva space-filling di Hilbert e si assegna la risorsa al primo nodo sull'anello che possiede un ID uguale o maggiore del valore della chiave surrogata appena calcolata. La funzione di Hilbert tramite un procedimento ricorsivo, identifica ad ogni iterazione un insieme di celle. I punti centrali di tali celle sono uniti dalla curva di Hilbert (fig. 2.4).

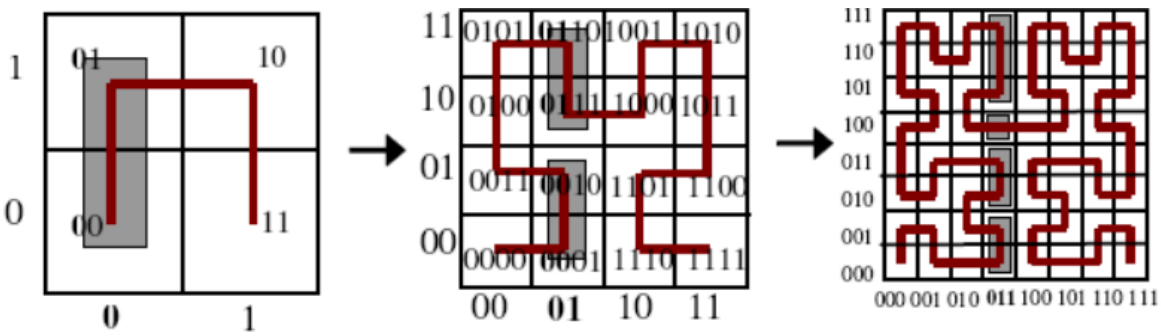


Figura 2.4 – Squid: Funzione di Hilbert

Si definiscono *cluster* un insieme di celle generate ad una specifica iterazione della curva di Hilbert. In Squid l'elaborazione di una query prevede l'esecuzione di due passi distinti. Il primo passo prevede la traduzione della query in un insieme di *cluster rilevanti* e il secondo nell'interrogazione dei nodi corrispondenti (fig. 2.5). Un cluster è definito rilevante se almeno un punto al proprio interno appartiene all'insieme delle risorse mappate sull'anello Chord.

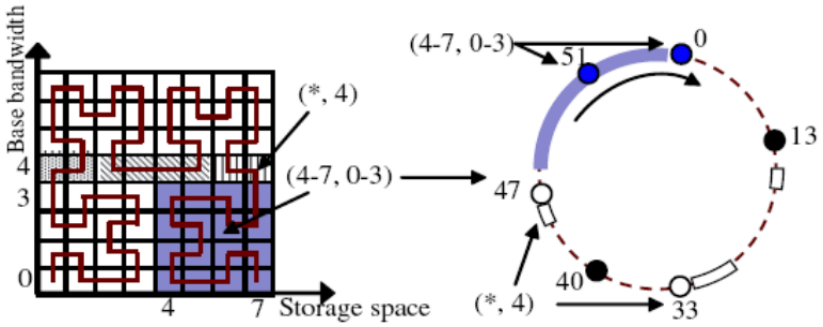


Figura 2.5 – Squid clusters

In Squid, per evitare che un numero elevato di cluster possa rendere la soluzione non scalabile, è stato introdotto un ulteriore passo di ottimizzazione: la generazione dei cluster rilevanti avviene in maniera incrementale e guidata dai dati. Essendo ogni cluster identificato da un prefisso nell'albero, per poter ottimizzare il processo di ricerca, i cluster vengono raffinati incrementalmente dai soli nodi, che ricadono all'interno dei cluster via via raffinati (fig. 2.6).

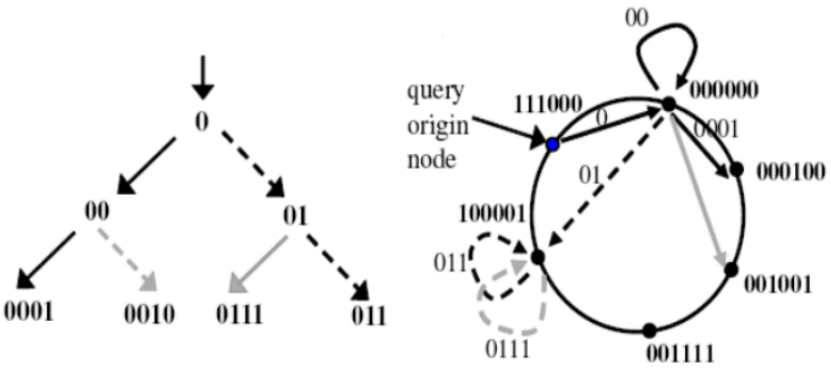


Figura 2.6 – Raffinazione dei clusters in Squid

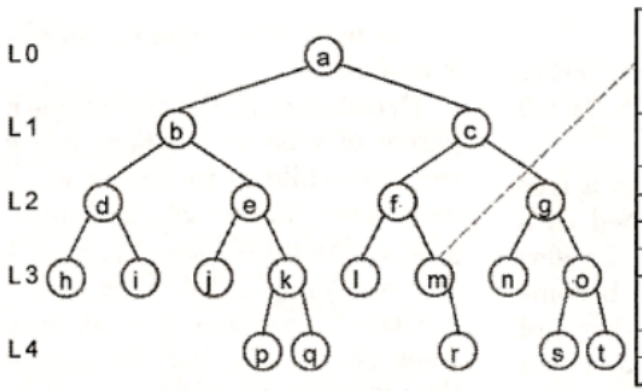
2.4.3 Baton

Baton, BAlanced Tree Overlay Network [17] è una rete P2P di tipo strutturato in grado di supportare range query monodimensionali mediante l'utilizzo di una struttura dati distribuita simile ad un *B-albero*. Grazie a tale struttura distribuita ad albero, Baton permette di effettuare query sia del tipo *Exact Match* che per *range*.

Ogni nodo dell'albero rappresenta esattamente un peer della rete e mantiene una parte della struttura dati distribuita. Ciascun nodo Baton è identificato tramite il proprio indirizzo IP e possiede come proprio stato interno, una routing table (fig. 2.7), che contiene il riferimento al nodo padre, i riferimenti ai nodi figli, ad un insieme di nodi adiacenti secondo una visita simmetrica o *in order* dell'albero e ad m nodi, vicini dello stesso livello. In particolare gli m nodi vicini a destra e a sinistra del nodo sono selezionati secondo la successione

$$m - 2^2, m - 2^1, m - 2^0, m + 2^0, m + 2^1, m + 2^2$$

supponendo una numerazione dei nodi per livelli successivi a partire dal livello zero e da sinistra verso destra.



Node m: level=3, number=6
parent=f, leftchild=null, rightchild=r
leftadjacent=f, rightadjacent=r
Left routing table:

	Node	Left child	Right child	Lower bound	Upper bound
0	l	null	null	l_{lower}	l_{upper}
1	k	p	q	k_{lower}	k_{upper}
2	i	null	null	i_{lower}	i_{upper}

Right routing table:

	Node	Left child	Right child	Lower bound	Upper bound
0	n	null	null	n_{lower}	n_{upper}
1	o	s	t	o_{lower}	o_{upper}

Figura 2.7 – Routing table di un nodo in Baton

Baton, per poter effettuare delle ricerche, utilizza, oltre all'albero bilanciato, un'ulteriore struttura dati ad indici distribuita. Ciascun nodo si occupa di gestire un intervallo di valori, definito dagli estremi *lower* e *up*. L'assegnazione dell'intervallo di valori, che ogni nodo deve gestire, è stabilito con il seguente criterio: il valore *lower* assume il

valore massimo contenuto nel sottoalbero sinistro del nodo, mentre *up* assume il valore minimo presente nel sottoalbero destro (fig. 2.8).

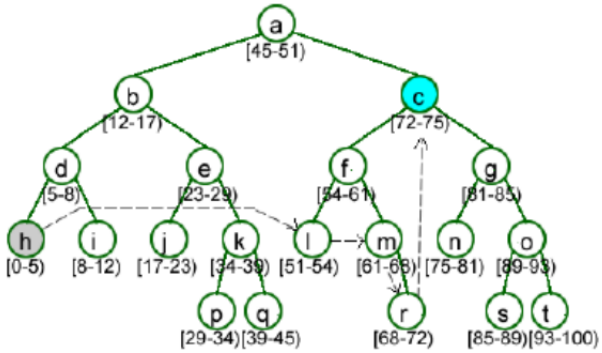


Figura 2.8 – Indici distribuiti in Baton

In fig. 2.8 è possibile notare come l'assegnazione ai nodi foglia degli intervalli di valori da gestire sia perfettamente ordinata e come, a differenza dei B^+-alberi, anche i nodi interni dell'albero si occupino di gestire loro stessi un range di valori.

Attraverso l'esempio riportato in figura 2.8 mostriamo adesso il processo di ricerca. Supponiamo che il nodo h voglia cercare un valore gestito dal nodo c. L'algoritmo di ricerca, illustrato in [17], effettua i seguenti passaggi. Il nodo h confronta il valore cercato con il proprio range, determinando, attraverso la sua routing table, che il valore di *up* risulta inferiore al valore cercato e dunque inoltra la query al vicino destro più estremo, tale per cui il relativo limite di *lower* risulti inferiore al valore cercato. Ricordiamo che i vicini sono memorizzati esponenzialmente sempre più distanti. Il nodo l effettua un procedimento analogo e inoltra la query al vicino m, il quale, non avendo nella propria routing table nessun vicino destro in grado di soddisfare la condizione di limite inferiore, invia la query al figlio destro r. Il nodo r, non trovando nessun vicino destro e non possedendo nessun figlio invia la query ad un nodo fisicamente adiacente, c, concludendo in questo modo l'operazione di ricerca.

Nel caso di range query, l'algoritmo si comporta in maniera del tutto analoga, intersecando i range esaminati. Dall'esempio proposto possiamo notare come l'operazione di ricerca sia limitata nel numero di nodi da visitare. Infatti, come mostrato dagli autori, il costo di ricerca al caso pessimo risulta logaritmico e pari alla profondità dell'albero; inoltre si ha la certezza di non ottenere risultati falsi negativi.

L'operazione di *join* in Baton è separata in due fasi. Nella prima fase viene determinata la collocazione del nuovo nodo all'interno dell'albero e nella seconda vengono eseguite le operazioni di join vere e proprie. Il costo dell'operazione di join è determinato dalla prima fase di ricerca. Gli autori mostrano in [17] come tale ricerca non coinvolga più di $O(logN)$ nodi, con N pari al numero di nodi o peer nella rete.

Mostriamo adesso come si comporta Baton nella gestione degli aggiornamenti dei valori ed in particolare di come possano innescare il bilanciamento dell'albero. Ipotizziamo che il valore gestito da un nodo abbia una variazione oltre i limiti del range gestito localmente e, considerando l'esempio in figura 2.9, supponiamo che l'aggiornamento comporti lo spostamento di un valore dal generico nodo *e* al nodo *g;* questo spostamento può essere determinato attraverso una semplice procedura di ricerca. Supponiamo che il nodo *g* adesso si trovi a gestire un elevato numero di valori ed occorra procedere ad una suddivisione del range da lui gestito ed eventualmente ad un bilanciamento dei nodi. Il caso riportato in figura 2.9 (a) mostra esattamente la situazione in cui è presente uno sbilanciamento del carico nel nodo *g*.

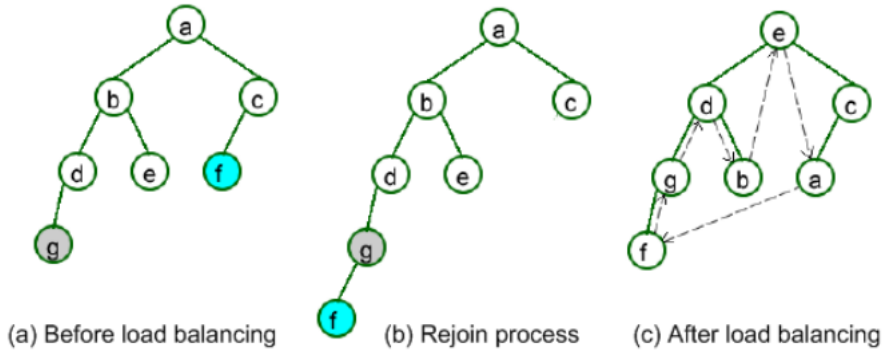


Figura 2.9 – Esempio di bilanciamento del carico dei nodi in Baton

L'algoritmo di bilanciamento del carico procede nel seguente modo. Il nodo *g* identifica un nodo scarico presente in rete, per esempio il nodo *f,* il quale delega la gestione del proprio range di valori al nodo padre *c* ed effettua una nuova join sull'albero posizionandosi come nodo figlio di *g;* a questo punto il nodo *g* divide il proprio intervallo di valori da gestire con *f* (fig. 2.9 (b)). In seguito al bilanciamento del carico l'esempio in figura mostra come sia necessario anche un bilanciamento dell'albero. La procedura è simile a quella utilizzata negli alberi AVL. I movimenti effettuati sono illustrati in

figura 2.9 (c) tramite la linea tratteggiata. Il nodo *f* rimpiazza il nodo *g*, il quale a sua volta rimpiazza il nodo *d;* il nodo *d* rimpiazza il nodo *b*, *b* rimpiazza il nodo *e*, *e* il nodo *a* ed infine *a* assume la posizione originale avuta da *f*.

Il problema del bilanciamento del carico si rivela il punto critico della proposta di Baton; infatti anche se il range di valori assegnati ad un nodo non varia, il numero di query ricevute da un nodo potrebbe essere signicativamente differente da quelle ricevute da altri nodi. Pertanto anche in questo caso il ribilanciamento dinamico della struttura è necessario. Si noti inoltre che tale bilanciamento del carico può comportare, nel caso peggiore, un trasferimento totale dei valori del nodo carico al nodo scarico e coinvolgere potenzialmente tutti i nodi dell'albero. Questa situazione potrebbe ricrearsi frequentemente in presenza di attributi altamente dinamici.

2.4.4 RST: Range Search Tree

La struttura dati *Range Search Tree (RST)* è la soluzione proposta in [18]. RST è una struttura basata sul mantenimento di un albero distribuito tra i peer della rete ed in grado di fornire supporto a range query multidimensionali. Per illustrare l'albero RST, occorre prima descrivere la tecnica di base utilizzata per effettuare il bilanciamento del carico tra i nodi. Una risorsa R è definita da un insieme di coppie attributo/valore della forma

$$R = \{(a_1, v_1), (a_2, v_2), ..., (a_n, v_n)\}$$

Ad ogni coppia (a_i, v_i) è associata una matrice di bilanciamento del carico denominata *Load Balancing Matrix* o *LBM*. In questa matrice è tenuta traccia del carico relativo alle registrazioni e alle richieste effettuate per la coppia (a_i, v_i) ed in particolare serve ad organizzare i peer già presenti in rete, per condividerne il carico.

In figura 2.10 è mostrata la matrice di bilanciamento per l'attributo (a_i, v_i). I nodi nelle colonne memorizzano una partizione della risorsa R, ovvero un sotto-insieme delle coppie attributo-valore, di cui è composta la risorsa con il vincolo che sia sempre inclusa la coppia (a_i, v_i). I nodi di una stessa colonna consistono invece di repliche della stessa partizione. La matrice *LBM* aumenta o diminuisce di dimensione dinamicamente nel tempo, a seconda del carico di registrazioni o interrogazioni effettuate nella rete. Per tenere traccia della dimensione della LBM si ricorre ad un *head node,* il quale memorizza il numero di partizioni e repliche.

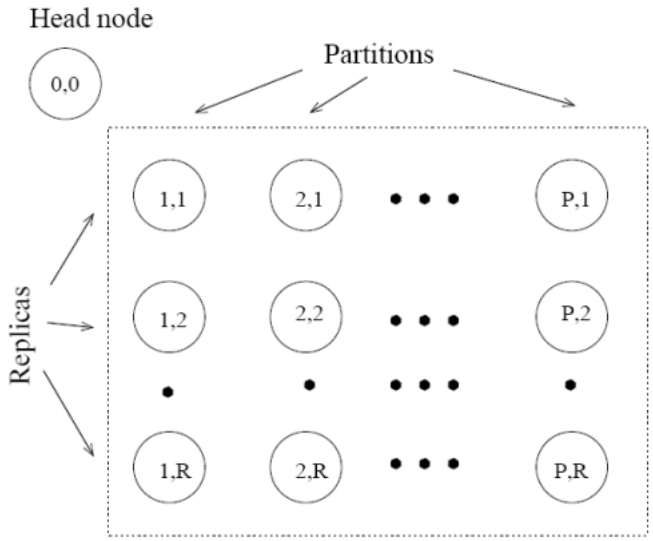


Figura 2.10 – Matrice di bilanciamento del carico (LBM) per (a_i, v_i)

Descriviamo adesso come il generico nodo memorizzi una risorsa R attraverso l'uso di tale matrice. Per ogni coppia di valori della risorsa, (a_i, v_i), vengono recuperate attraverso l'*head node* il numero di partizioni e repliche. In seguito tramite l'ausilio di soglie del carico, mantenute localmente, viene decisa la partizione p, in cui memorizzare la risorsa R e viene applicata la seguente funzione per determinare tutti i nodi replica r della stessa partizione:

$$N \leftarrow H((a_i, v_i), p, r)$$

La matrice LBM appena descritta risulta alla base della struttura Range Search Tree.

Un albero RST è costruito sul dominio dei valori di un singolo attributo. Tra i nodi di uno stesso livello si effettua il partizionamento del dominio in intervalli di differente granularità. Tutti i nodi di livello l_i possiedono degli intervalli di ampiezza minore rispetto ai nodi di livello l_{i+1}, ma in ogni caso l'unione degli intervalli di un livello ricopre sempre l'intero dominio. Un esempio di partizionamento dell'albero è mostrato in figura 2.11(a); il dominio dei valori è definito nell'intervallo [0; 7].

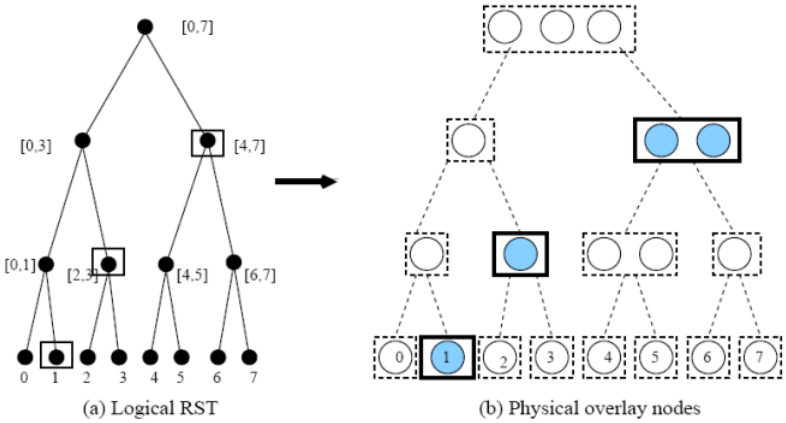


Figura 2.11 – Esempio struttura RST

Illustriamo adesso le operazioni di registrazione ed interrogazione nell'albero RST. Per memorizzare una risorsa si estende l'algoritmo descritto precedentemente con la matrice LBM. Considerando la generica coppia (a_i, v_i), il valore v_i identifica un percorso specifico all'interno dell'albero RST denominato *Path(v_i)*. La memorizzazione della risorsa R viene effettuata esclusivamente tra i nodi del Path. Recuperando i valori della LBM, l'algoritmo viene poi esteso determinando i nodi nel seguente modo:

$$N \leftarrow H((a_i, v_i), [s, e], p, r)$$

i parametri *[s, e]* rappresentano il range del nodo del path, in cui memorizzare i valori p ed r gli indici della relativa LBM (fig 2.11(b)). Questo procedimento di memorizzazione, come notato degli autori, risulta non ottimizzato in quanto l'attributo potrebbe essere memorizzato in un sottoinsieme dei nodi del Path. Idealmente i nodi in cui memorizzare, dovrebbero essere i nodi RST i cui range siano non eccessivamente granulari, ma neanche tali da concentrare le registrazioni verso pochi nodi producendo uno sbilanciamento del carico. Per ottenere questo comportamento, viene introdotto il concetto di *banda* dell'albero RST. La banda rappresenta un sottoinsieme dei nodi di un Path, determinati dinamicamente dalla rete, in cui è conveniente memorizzare le risorse. Le informazioni sulla banda vengono mantenute in maniera simile a quanto fatto con la matrice LBM attraverso l'head node e il *Path Maintenance Protocol (PMP)* (fig 2.12).

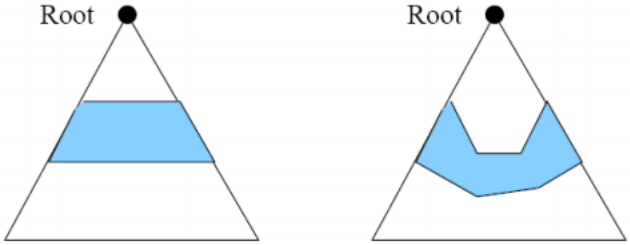


Figura 2.12 – Determinazione della banda

Introducendo questa ottimizzazione, la procedura di ricerca varia leggermente, dovendo in via preliminare determinare la banda e successivamente memorizzare esclusivamente nei nodi del Path all'interno della banda.

Illustriamo adesso il meccanismo di interrogazione. Nella richiesta di una range query occorre notare come esistano diversi modi per decomporre il range richiesto utilizzando l'albero RST e l'efficienza dell'algoritmo di query è dunque in parte legata alla determinazione di una decomposizione ottimale della query e quindi dei nodi RST da contattare. A guidare il processo di decomposizione viene introdotta una metrica chiamata *rilevanza*. Supponendo di avere una range query Q con intervallo *[s,e]*, l'algoritmo di query potrà decomporre l'intervallo originario in k sub-queries corrispondenti a k nodi dell'albero RST, $N_1, N_2, ..., N_k$. La *rilevanza* r è definita come

$$r = \frac{R_q}{\sum_{i=1}^{k} R_i}$$

dove R_i esprime l'ampiezza del range gestito dal nodo N_i e R_q l'ampiezza della query originaria. Intuitivamente la rilevanza indica quanto la decomposizione scelta corrisponda bene alla query originaria. Ulteriori dettagli sulla determinazione ottimale dei nodi RST possono essere trovati in [18]. A questo punto l'esecuzione di una range query si compone di tre passi: recupero preliminare delle informazioni sulla banda, decomposizione del range richiesto in una lista di nodi RST ed interrogazione dei vari nodi in accordo con la banda individuata e le informazioni contenute nella LBM.

2.4.5 Cone

Cone [19] è una rete P2P strutturata che utilizza una struttura dati ispirata ad un albero simile ad un heap. In Cone è possibile effettuare query del tipo *Trova k risorse*

di dimensione maggiore di S. Cone è costruito sopra un livello di routing che supporti la ricerca basata sui prefissi, come ad esempio Chord. L'albero Cone è quindi un albero binario dei prefissi in cui le foglie vengono assegnate ai peer in maniera del tutto casuale attraverso la funzione di hashing della DHT. Ogni nodo interno N possiede il valore massimo contenuto nel sottoalbero radicato in N, ottenuto mediante l'applicazione della *heap-property* (fig. 2.13). Un peer P mantiene una Routing Table, che memorizza una porzione consecutiva di nodi logici la quale corrisponde ad un percorso all'interno dell'albero. In particolare per ogni nodo logico N, non foglia, deve valere la seguente proprietà:

$$N = \begin{cases} \text{left}(N), \ \text{left}(N).\text{key} < \text{right}(N).\text{key} \\ \text{right}(N), \text{altrimenti} \end{cases}$$

Tale proprietà implica che il nodo N di livello $l > 0$ sia esso stesso anche uno dei nodi figli. Una volta collocato nell'albero Cone, il nodo non varierà la sua posizione durante tutta la permanenza in rete. Infatti la costruzione dell'albero Cone è determinata implicitamente dalle relazioni mantenute, distribuita tra tutti i nodi nelle *routing tables*. Una variazione del valore gestito comporterà una riorganizzazione delle tabelle di routing dei soli nodi coinvolti.

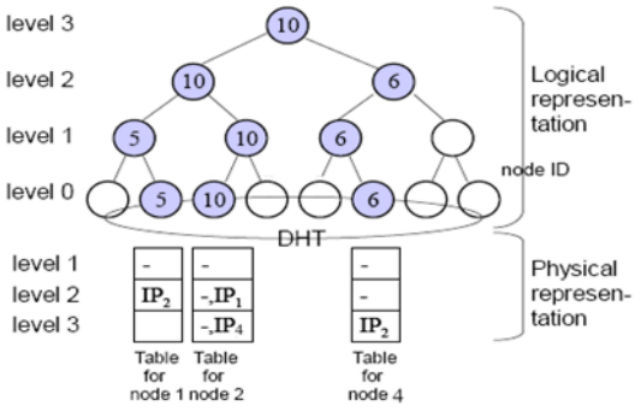


Figura 2.13 – Architettura Cone

La memorizzazione di un valore avviene localmente e, come mostrato dagli autori, in un numero di nodi logaritmico. Per effettuare delle ricerche in Cone, viene sfruttata la *heap-property*. La ricerca consiste in una risalita dell'albero attraverso l'utilizzo delle routing tables e sfrutta al massimo le operazioni di potatura dell'albero, per

ridurne l'esplorazione. Il costo dell'operazione è logaritmico. Cone, pur essendo una struttura ad albero, presenta delle buone proprieta di bilanciamento del carico. Per poter analizzare la struttura occorre introdurre i concetti di *Data Traffic* e *Control Traffic*.

Data Traffic. Si definisce *Data Traffic* il carico di query che ogni nodo dell'albero gestisce. Dato un insieme di peer $P_1, P_2, ..., P_z$ aventi chiavi all'interno di un range, si definisce $Prob_D(P_i)$ la probabilità che il nodo P_i soddisfi una query. Il carico risulta bilanciato se vale

$$\forall i, j \leq z, i \neq j \qquad Prob_D(P_i) = Prob_D(P_j)$$

Control Traffic. Si definisce *Control Traffic* il numero di messaggi di controllo scambiati dai nodi. Idealmente si ha bilanciamento del carico se il numero di messaggi scambiati dai nodi di tutto il sistema è identico. Formalmente, definita $Prob_C(P_i)$ la probabilità che per qualche query un messaggio di controllo venga inviato al nodo N_i, si ha bilanciamento del carico se vale

$$\forall i, j \leq z, i \neq j \qquad Prob_C(P_i) = Prob_C(P_j)$$

Analisi del data traffic

Iniziamo considerando due esempi in cui si verifica il maggiore sbilanciamento del carico, per poi passare ad un'analisi più generale. Supponiamo di cercare una risorsa che soddisfi una determinata query. Ipotizziamo che tale query possa essere risolta esclusivamente da due peer P_1, P_2 e che l'associazione tra i peer e i nodi dell'albero Cone sia quella mostrata in figura 2.14 (a). Il peer P_2 è il padre al nodo logico di livello uno del peer P_1. In questo caso si ha lo sbilanciamento massimo del carico e tutte le query originate dai T-1 peer saranno dirette a P_2 mentre a P_1 arriveranno le sole query originate da P_1 stesso. In questo caso lo sbilanciamento massimo è pari a $O(T)$, con T pari al numero di peer presenti nella rete. Come caso degenere si può avere la situazione illustrata in figura 2.14 (b), in cui i peer P_1 e P_2 abbiano ottenuto lo stesso identicativo dalla DHT e quindi siano stati assegnati alla stessa foglia. In questo caso P_1, ad esempio, risponderà alle query originate da se stesso, mentre lascerà a P_2 la gestione di tutte le altre. Lo sbilanciamento anche in questo caso risulta $O(T)$. Fortunatamente la probabilità che avvenga una tale configurazione per entrambi gli

scenari, risulta poco probabile. Infatti sia nel caso in cui i due nodi risultino adiacenti sia nel caso in cui abbiano generato lo stesso ID, la probabilità che questo accada è pari a $\frac{1}{2^h}$. Tale probabilità è calcolata assumendo che il numero di nodi fisici nella rete sia esattamente uguale al numero di foglie nell'albero; per valori inferiori di nodi la dimostrazione può essere effettuata in maniera analoga.

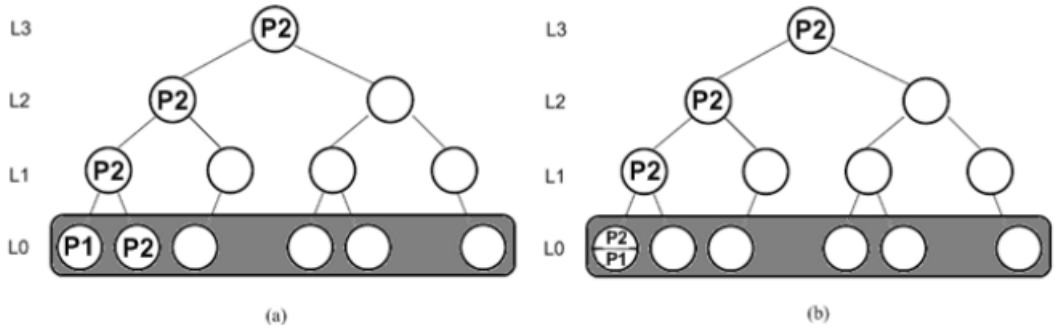


Figura 2.14 – Casi di massimo sbilanciamento dell'albero Cone

A questo punto possiamo analizzare il caso medio. Consideriamo il caso in cui P_1 sia adiacente a P_2 attraverso un sotto-albero di altezza k e che P_2 sia il padre dal livello k fino alla radice (fig. 2.15).

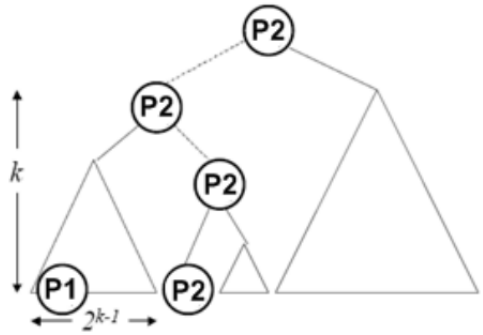


Figura 2.15 – Caso generico del Data Traffic in Cone

L'assegnamento del relativo ID ai nodi è effettuato dalla funzione hash della DHT e la probabilità che P_1 sia esattamente in un sotto-albero adiacente di altezza k è pari a $\frac{2^k}{2^h}$. Pertanto la probabilità che due nodi P_1 e P_2 diventino adiacenti in un sotto-albero

di altezza k è di $\frac{1}{2^{h-k}}$. Nello specifico P_2 diventerà il nodo padre di livello k. Tenendo conto di quanto detto possiamo definire le seguenti quantità:

- $Prob_D(P_1) = \frac{2^k}{2^h}$, la probabilità che il nodo P_1 soddisfi una determinata query.
- $Prob_D(P_2) = 1 - \frac{2^k}{2^h}$, la probabilità che il nodo P_2 soddisfi una determinata query

Il rapporto tra i due Data Traffic risulta $r = \frac{\text{Prob}_D(P_1)}{\text{Prob}_D(P_2)} = \frac{2^h - 2^k}{2^k}$. Fissato k, lo sbilanciamento dell'albero risulta essere pari a

$$\left(\frac{1}{2^h - 2^k}\right) \times \left(\frac{2^h - 2^k}{2^k}\right) = \frac{2^h - 2^k}{2^h}$$

Quindi per calcolare lo sbilanciamento al caso generico occorre sommare per tutti i possibili valori di k:

$$\frac{2^h-1}{2^h} + \sum_{k=0}^{h-1} \frac{(2^h - 2^k)}{2^h} = \frac{2^h-1}{2^h} + \frac{1}{2^h}((2^h-1)+(2^h-2)+...+(2^h-2^{h-1}) = 1-2^h+h+\sum_{k=1}^{n}\frac{1}{2^h} = h$$

Ottenendo quindi che nel caso medio, per la ricerca di una risorsa, il rapporto tra il peer selezionato il maggior numero di volte e quello selezionato il minor numero risulta non superiore a $h = log(T)$.

Analisi del Control Traffic

Per poter effettuare l'analisi del Control Traffic è necessario fare la seguente ipotesi: la distribuzione delle chiavi possedute dai peer è identica alla distribuzione delle query effettuate, ed entrambe le distribuzioni sono continue. Questa ipotesi risulta necessaria per poter sfruttare le proprietà di simmetria e mettere in relazione la distribuzione delle chiavi con la distribuzione delle query. Se si considera l'esempio in figura 2.15, la probabilità che una query originata dal nodo P_1 arrivi al nodo P_2, situato a livello k, è uguale alla probabilità che il valore massimo cercato dalla query sia il più grande valore tra tutti i valori gestiti nei 2^{k-1} peer del sotto-albero. Questo equivale a prelevare tra i $2^{k-1} + 1$ campioni dalla distribuzione delle query (l'uno in più rappresenta la query) e stimare la probabilità che il valore richiesto sia esattamente il più grande valore tra tutti quelli gestiti nel sotto-albero. Dato che ogni elemento del campione, per ipotesi di simmetria, possa essere il valore massimo con uguale probabilità, allora la probabilità di selezionare il valore massimo risulta di $\frac{1}{2^{k-1}+1}$. Ritornando all'esempio in figura 2.15, il numero di nodi dal quale la query può arrivare a P_2 è pari a 2^{k-1}. L'indice $k-1$

è dato dal fatto che in Cone, un peer che gestisce il nodo P_2, deve necessariamente gestire anche uno dei suoi figli, nell'esempio il figlio destro. Il carico complessivo di query che potrà raggiungere P_2 è quello proveniente dal solo sotto-albero di livello $k-1$; quindi i messaggi di controllo scambiati a seguito di un'interrogazione sono pari a $2^{k-1}\, x\, \frac{1}{2^{k-1}+1}$. Possiamo concludere che al caso pessimo il fattore di massimo sbilanciamento del Control Traffic è rappresentato dal caso in cui P_2 sia situato nella radice dell'albero Cone, quindi:

$$\sum_{k=1}^{h} \left(\frac{2^{k-1}}{2^{k-1}+1} \right) < h = \log(T)$$

2.4.6 Confronto soluzioni presenti in letteratura

Le proposte illustrate presentano degli indubbi vantaggi in termini di aumento di espressività nel linguaggio d'interrogazione, ma occorre prestare attenzione agli elementi caratterizzanti e agli aspetti critici.

Nel caso delle soluzioni basate sulla manipolazione delle funzioni hash, come MAAN [15], la scelta della funzione hash risulta il fulcro centrale del modello prestazionale, che, nel caso di distribuzioni dei dati non uniformi, è estremamente critico. A questo aspetto si aggiuge l'ulteriore overhead che MAAN possiede nella presenza di risorse dinamiche, dovendo memorizzare tutti gli attributi della risorsa.

Nell'approccio presentato da Squid [16], le risorse sono state preventivamente trasformate da una funzione di mapping. Come al caso precedente, Squid, memorizzando le risorse direttamente sull'anello Chord, risulta essere sensibile alle distribuzioni non uniformi. In aggiunta si ha anche la presenza della nota problematica *curse of dimensionality,* che rende critica la gestione della struttura dati al crescere del numero di dimensioni dello spazio originario.

Infine negli ultimi approcci (Baton [17], RST [18] e Cone [19]) si è indagato su strutture basate sulla costruzione di una rete strutturata ispirata agli alberi. Queste soluzioni presentano degli aspetti fondamentali da analizzare:

- bilanciamento del carico
- gestione degli attributi dinamici
- overhead di gestione

Nello specifico, l'approccio Baton mostra i suoi limiti sia con la presenza di attributi dinamici che con un elevato overhead di gestione in caso di bilanciamento della struttura; infatti in Baton la variazione di un attributo può determinare uno sbilanciamento dei nodi e quindi innescare delle procedure di riorganizzazione della struttura.

Nella struttura RST i limiti maggiori si hanno sia nella scarsa scalabilità del sistema, basato nel recupero di informazioni essenziali attraverso *head node* che negli elevati costi di mantenimento dinamico delle LBM e banda.

L'ultimo approccio Cone risulta molto promettente, sia per il bilanciamento del carico sfruttando le proprietà di distribuzione dei dati attraverso funzione hash, che nel mantenimento della struttura a seguito dei cambiamenti dei valori. Per contro, Cone presenta una scarsa espressività sulle query effettuabili.

Lo studio della letteratura ha quindi mostrato come ogni proposta presenti soluzioni sub-ottime al problema del *Resource Discovery* diffcili da coniugare. La prossima soluzione che verrà riportata, *HASP, Hierarchical Aggregation over Space-filling,* si basa su XCone [20] [44] e pone proprio l'accento sulla gestione degli attributi dinamici e di come possano essere interrogati attraverso range query. XCone generalizza Cone, il quale offre una gestione efficace degli attributi dinamici, attraverso il supporto di un linguaggio d'interrogazione più espressivo quali le range query. *HASP* si propone di estendere XCone al fine supportare le range query multidimensionali.

Capitolo 3

Le curve space-filling

Le curve space-filling sono curve continue che permettono la linearizzazione di uno spazio n-dimensionale in modo univoco. In questo capitolo verranno prima presentate le caratteristiche generali delle curve space-filling nella sezione 3.1 e successivamente verranno descritte in dettaglio alcune di queste curve nelle sezioni dalla 3.2 alla 3.5. Infine nella sezione 3.6 verranno analizzate le proprietà di clustering delle varie curve descritte in precedenza.

3.1 Le curve space-filling: caratteristiche generali

Il concetto di *curva space-filling* nasce nel diciannovesimo secolo e viene originariamente attribuito al matematico *Giuseppe Peano* (1858-1932), il quale fornì il primo esempio di *una curva che riempie lo spazio: la curva di Peano*. Peano fu il primo ad esprimere tale curva in termini matematici e rappresentò le coordinate dei punti nello spazio tramite una radice ternaria. La prima rappresentazione grafica, o geometrica, di una curva space-filling è attribuita al matematico tedesco *David Hilbert* (1862-1943), il quale espresse il concetto di curva space-filling in due dimensioni tramite una rappresentazione binaria delle coordinate dei punti nello spazio bidimensionale.

Il concetto di curva space-filling può essere espresso in due modi. Una curva può essere definita come l'immagine dell'intervallo unitario definito da una funzione continua. Una curva di tipo space-filling rappresenta l'immagine di un particolare tipo di funzione suriettiva il cui codominio è dato dal quadrato unitario $[0;1]^2$o cubo $[0;1]^3$ o

in generale iper-cubo $[0; 1]^n$ con n pari al numero di dimensioni dello spazio [22].

Il limite di questa funzione definisce un mapping dall'intervallo unitario al suo prodotto cartesiano. Visto in un'altra prospettiva, le curve space-filling sono curve continue che permettono di mappare tutti i punti appartenenti ad uno spazio $[0,\ 1]^n$ su un intervallo unitario [0, 1]. In particolare tale famiglia di curve permette il mapping da uno spazio n-dimensionale ad uno spazio uni-dimensionale o lineare in modo univoco, in quanto per ogni punto dello spazio n-dimensionale, definito da n attributi, esiste un mapping verso uno ed un solo punto nello spazio uni-dimensionale chiamato *chiave derivata* o *chiave surrogata*.

Ciascun punto dello spazio n-dimensionale può rappresentare un qualsiasi tipo di risorsa definita dal valore di n attributi ed essere trasformato in un punto nello spazio uni-dimensionale.

Uno studio dettagliato delle curve space-filling dal punto di vista matematico può essere trovato in [29], dal quale riprendiamo la seguente definizione:

Definizione 3.1.1. *Dato un intervallo unitario I ed uno spazio euclideo n-dimensionale E^n con $n \geqslant 2$, se una funzione continua $f : I \to E^n$ possiede un'immagine con un valore positivo per la misura di Peano-Jordan, allora l'immagine di tale funzione è una curva space-filling.*

La misura di Peano-Jordan, chiamata in inglese *Jordan content,* è un'estensione del concetto di grandezza applicata a oggetti geometrici di forma complessa.

L'utilizzo delle curve space-filling si rivela interessante in molti contesti applicativi in quanto, nel caso si voglia estendere un'applicazione esistente per il caso unidimensionale al caso multidimensionale, è possibile riutilizzare gli algoritmi sviluppati per attributi di tipo lineare sfruttando l'algoritmo di mapping dagli n attributi nello spazio n-dimensionale alla chiave derivata nello spazio unidimensionale.

Esistono molti tipi di curve space-filling, ognuna caratterizzata da un diverso algoritmo per il calcolo della chiave derivata e di conseguenza da un diverso mapping dei punti n-dimensionali nello spazio uni-dimensionale. Le più note sono la *curva di Peano*, la *curva di Hilbert*, la *Gray-curve*, la *Z-order curve* [29].

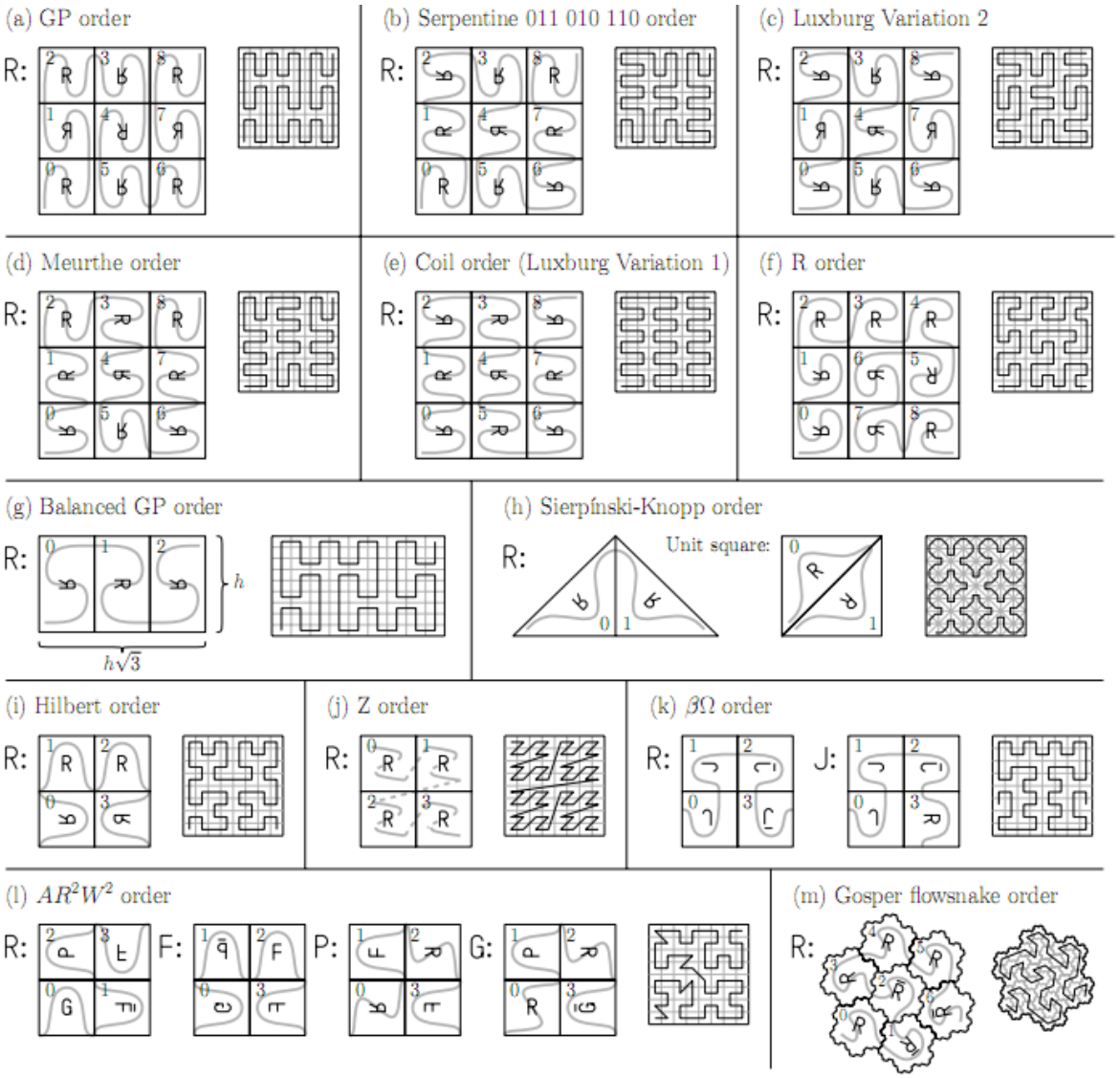


Figura 3.1 – Definizione di alcune curve space-filling ed esempio delle polilinee che le approssimano

Le caratteristiche principali di una curva space-filling sono:

1. *ciascun punto dello spazio n-dimensionale possiede un'immagine nello spazio uni-dimensionale chiamata chiave derivata*
2. *chiavi derivate vicine nello spazio uni-dimensionale appartengono a punti dell'iper-spazio n-dimensionale vicini tra loro*

3. *la curva è continua*

Una proprietà valida solo per alcune curve space-filling è la seguente:

- *ciascun punto dello spazio n-dimensionale è visitato dalla curva space-filling una ed una sola volta* (valida solo per alcuni tipi di *SFC*)

La seconda caratteristica delle curve space-filling è valida solamente nel verso in cui è espressa, in quanto non è sempre vero che a punti vicini nello spazio n-dimensionale corrispondono chiavi derivate nello spazio uni-dimensionale di valore vicino tra loro. Le curve space-filling si possono quindi classificare a seconda della loro capacità di mantenere la località, cioè di mappare punti vicini nello spazio n-dimensionale in punti vicini sulla curva. Occorre inoltre notare che esiste una classe di curve che non soddisfano la proprietà 2, ma che spesso vengono classificate ugualmente come cuve space-filling. Queste curve vengono dette *discontinue.* Si parla di discontinuità quando una coppia di punti non adiacenti nello spazio sono consecutivi nel valore delle chiavi derivate.

Una di queste curve è la *curva Z-order*. Considerando la curva Z-order descritta in figura 3.2, si possono notare due cose:

1. non sempre a punti vicini nello spazio n-dimensionale corrispondono nello spazio uni-dimensionale chiavi derivate con valori vicini tra loro. Ad esempio il punto di coordinate (3;1) è vicino topologicamente al punto (4;1), ma le loro chiavi derivate possiedono rispettivamente i valori 7 e 18 non vicini tra loro.

2. la curva è definita discontinua poichè a punti contigui sulla curva possono corrispondere punti nello spazio n-dimensionale che non sono contigui. Si consideri ad esempio il punto 15 ed il punto 16. Tali punti sono consecutivi sulla curva, ma corrispondono a punti non vicini nello spazio n-dimensionale.

I parametri principali che definiscono una curva space-filling sono due:

- la dimensione n
- l'ordine di raffinamento k

Il primo parametro, la *dimensione*, esprime il numero degli attributi i cui valori definiscono ciascun punto nell'iper-spazio n-dimensionale, mentre il secondo, l'*ordine*, definisce il range in cui sono contenuti i valori di ciascuno degli n attributi, $[0\,;2^k-1]$. Se la curva space-filling possiede un ordine di raffinamento maggiore, essa attraverserà un numero maggiore di punti nello spazio n-dimensionale e di conseguenza sarà definito un numero maggiore di chiavi derivate nello spazio uni-dimensionale.

In figura 3.3 è possibile osservare una particolare curva space-filling, la curva di Hilbert, a diversi ordini o livelli di raffinamento.

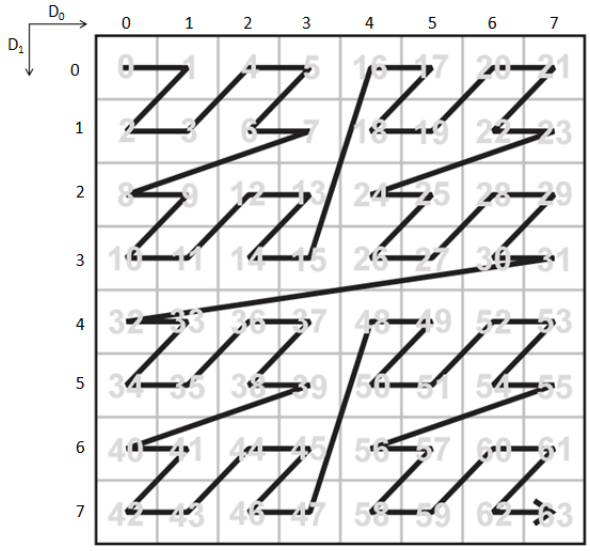


Figura 3.2 – Mapping delle chiavi derivate sulla Z-Curve di dimensione $n = 2$ e ordine $k = 3$

Quindi un'approssimazione al livello k di una curva space-filling per uno spazio n-dimensionale mappa un intero appartenente all'intervallo $[0\,;2^k-1]$ in uno spazio n-dimensionale $[0\,;2^k-1]^n$. Il numero di chiavi derivate su cui viene eseguito il mapping di una curva space-filling di dimensione n e ordine k è indipendente dalla specifica curva space-filling ed è sempre pari a $2^{n\times k}$. Nella tabella 3.1.1 sono riportati alcuni esempi del numero di chiavi derivate di una curva appartenente alla famiglia delle curve space-filling, al variare del numero delle dimensioni n e dell'ordine k. E' importante osservare che per valori relativamente bassi di n e k si ottiene un numero di chiavi molto elevato. Come vedremo nel capitolo 5, questa caratteristica ha un forte impatto sugli algoritmi e le strutture dati definite in *HASP* in quanto le applicazioni considerate utilizzano

un numero di attributi ed un range di valori per attributo che portano alla definizione di uno spazio di dimensione molto elevata.

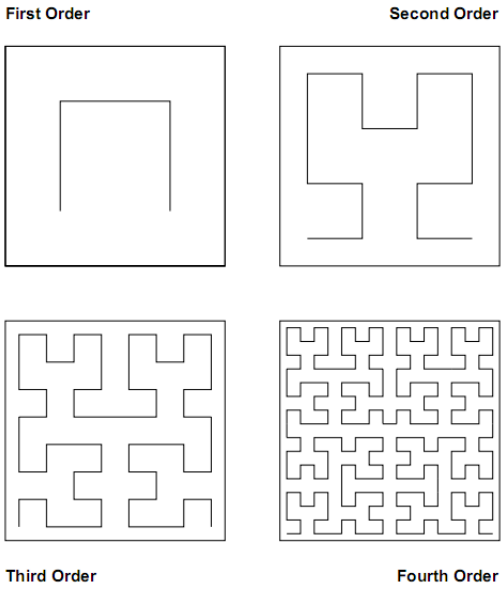


Figura 3.3 – Esempio della curva space-filling di Hilbert a diversi ordini di raffinamento

Tabella 3.1.1 – Numero di chiavi derivate al variare del numero di dimensioni *n* e dell'ordine *k*

	$k=1$	$k=2$	$k=3$	$k=5$	$k=7$	$k=9$
$n=2$	4	16	64	1024	2^{14}	2^{18}
$n=3$	8	64	512	2^{15}	2^{21}	2^{27}
$n=5$	32	1024	2^{15}	2^{25}	2^{35}	2^{45}
$n=6$	64	4096	2^{18}	2^{30}	2^{42}	2^{54}
$n=8$	256	2^{16}	2^{24}	2^{40}	2^{56}	2^{72}
$n=10$	1024	2^{20}	2^{30}	2^{50}	2^{70}	2^{90}

Le funzioni che descrivono le curve space-filling sono sempre suriettive, ma non necessariamente iniettive, in quanto possono visitare un punto nello spazio n-dimensionale più di una volta. Esempi di questo tipo di curve space-filling sono la *curva C(w) di Peano* descritta da *Moore* in [31], la curva di *Sierpinski* [29] e la *Dragon curve* [32]. In figura 3.4 sono mostrati due ordini differenti, il primo e il settimo, della la curva Dragon.

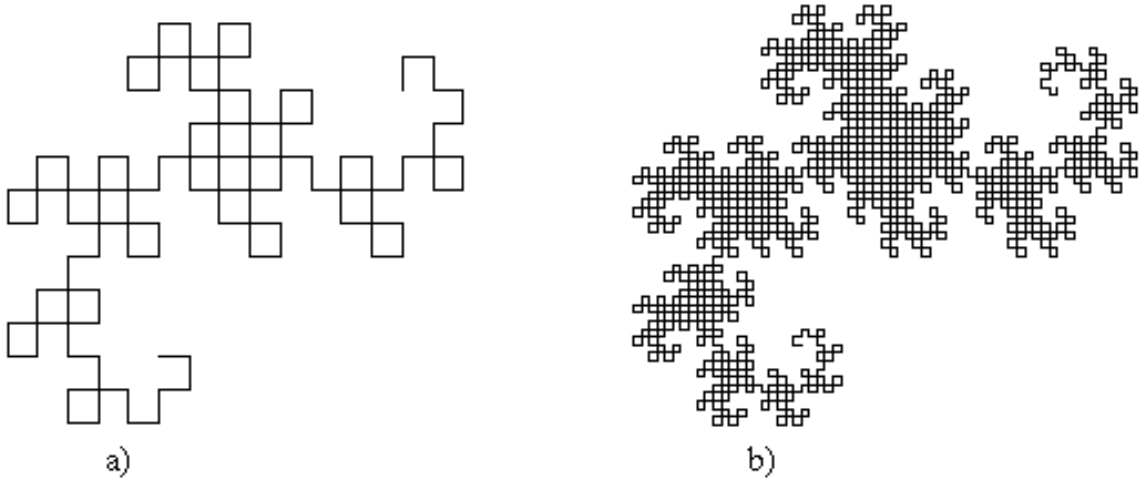


Figura 3.4 – la curva Dragon a) di ordine 1 e b) di ordine 7

Le ragioni principali per considerare l'utilizzo di curve di questo tipo sono principalmente due:

1. possono essere utilizzati algoritmi differenti per il calcolo del mapping e per l'esecuzione delle query, nonostante non siano realmente curve space-filling e molti di questi algoritmi sono più efficienti di quelli richiesti per la curva di Hilbert.

2. il grado di clusterizzazione delle chiavi derivate è differente, ma per alcune curve non troppo inferiore rispetto a quello della curva di Hilbert.

Le curve space-filling costituiscono un sottoinsieme di quelli che sono comunemente chiamati *frattali* [25], ma quest'ultimi non sono necessariamente una curva space-filling. Le curve space-filling, come detto, appartengono alla famiglia dei frattali in

quanto come essi possiedono la proprietà di auto-similarità (*self-similarity*), ovvero sono oggetti geometrici che si ripetono nella loro struttura allo stesso modo anche se visti su scale differenti [28]. A qualunque scala si osservi, l'oggetto presenta sempre gli stessi caratteri globali. Diminuendo la scala di un frattale tramite un ingrandimento della figura geometrica, si otterranno forme ricorrenti e compariranno nuovi dettagli ad ogni ingrandimento invece di ottenere una perdita di dettaglio come accadrebbe per qualsiasi altra figura geometrica che non è un frattale; nei frattali infatti ad ogni ingrandimento la figura si arrichisce di nuovi particolari. A differenza di un oggetto geometrico euclideo, un frattale non è definito tramite un'equazione, ma tramite un algoritmo ricorsivo; il procedimento di costruzione della curva può essere iterato, in linea teorica, un numero infinito di volte e ad ogni iterazione si ottiene una convergenza della curva per approssimazione verso il risultato finale. Il numero di iterazioni dell'algoritmo di definizione di un frattale è solitamente limitato in quanto, dopo un certo numero di iterazioni, l'occhio umano non è più in grado di distinguere le differenze tra due iterazioni successive e quindi il procedimento di costruzione del frattale può essere terminato dopo un certo numero di iterazioni. Alla base della proprietà di auto-similarità dei frattali vi è una particolare trasformazione geometrica chiamata *omotetia interna*, che permette di ingrandire o ridurre una figura lasciandone inalterata la forma.

Per dare una definizione di frattale, consideriamo un insieme N di trasformazioni del piano cartesiano

$$\{T_1, T_2, ..., T_N\}$$

ed applichiamolo ad un sotto-insieme A del piano in modo da ottenere una famiglia di N sottoinsiemi del piano

$$\{T_1(A), T_2(A), ..., T_N(A)\}$$

e chiamiamo A_1l'insieme ottenuto come unione di questi sotto-insiemi. Applicando nuovamente le N trasformazioni del piano all'insieme A_1, otteniamo un nuovo sotto-insieme A_2 dato dall'unione degli N insiemi immagine; continuando ad applicare tale procedimento otterremo una successione di insiemi

$$\{A_1, A_2, A_3...., A_i, ...\}$$

Questa successione di insiemi, determinata dalle iterazioni dell'algoritmo di generazione del frattale, sotto certe condizioni, converge verso un insieme limite F definito frattale *IFS (Iterated Function System)* [26], a partire dal quale non sarà più possbile notare cambiamenti apprezzabili della struttura del frattale. Un frattale *IFS* è un frattale ottenuto iterando un insieme di trasformazioni del piano.

Un tipo diverso di frattali sono i cosiddetti frattali *LS, Lindenmayer-System* [27] dal nome del suo scopritore, che rappresentano una generalizzazione di quelli *IFS*, infatti è possibile affermare che tutti i frattali *IFS* sono anche di tipo *LS*, ma non è vero il viceversa. Un frattale di tipo *LS* e non *IFS* si ha quando non è possibile suddividere la figura in un numero di copie simili alla figura stessa intera, per cui non vale la proprietà di auto-similarità, ma è altresì possibile dividere la figura in un certo numero di copie di un frattale *IFS*. Si può definire un frattale *LS* come una figura che si può dividere in un numero finito di frattali *IFS* e che dunque risulta non auto-simile a sè stessa ma auto-simile al suo interno.

In seguito saranno descritte in dettaglio la curva space-filling di Hilbert che appartiene alle vere curve space-filling e la curva Z-order che tecnicamente non è una vera e propria curva space-filling, ma possiede alcune proprietà interessanti.

3.2 La curva di Hilbert

Nel 1891, *David Hilbert* pubblicò un articolo nel quale illustrava il concetto di una curva space-filling in due dimensioni fornendo un metodo per la costruzione tramite iterazioni successive di una sequenza infinita di curve finite, ognuna delle quali definita da un ordinamento lineare dei sotto-spazi risultanti dal processo di costruzione. Hilbert mostrò inoltre che tale processo possiede, come limite, una curva che passa per tutti i punti dello spazio, è continua e non derivabile: *la curva di Hilbert*.

La costruzione della curva di Hilbert, rispetto a quella della curva Z-order, non può essere ottenuta applicando un certo numero di trasformazioni geometriche, in quanto, come è possibile osservare dalla figura 3.7, i passi della costruzione non sono autosimili. Per ottenere la curva non si può dunque ricorrere alla tecnica degli *IFS*, ma a quella degli *LS (Lindenmaier-System)*. Di seguito sono riportati i primi due passi del processo di costruzione della curva di Hilbert e successivamente la regola generale

per la costruzione della curva di Hilbert di ordine maggiore.

Passo 1. Il primo passo del processo di costruzione della curva di Hilbert prevede la divisione sia del quadrato $[0;1]^2$ che dell'intervallo lineare $[0;1]$ in quattro parti di uguale dimensione. Ciascun sotto-intervallo è mappato su un sotto-quadrante differente in modo tale che *sotto-quadranti corrispondenti a sotto-intervalli adiacenti possiedano un lato in comune*. Si osservi che questo mapping può portare ad ordinamenti diversi dei quadranti, corrispondenti a diversi orientamenti della curva del primo ordine. Questo curve, ruotate o riflesse, vengono utilizzate nella generazione della curva di ordine superiore. I sotto-quadranti sono dunque ordinati tracciando graficamente una linea, formata da segmenti rettilinei che uniscono i loro punti centrali nell'ordine indicato dai sotto-intervalli nello spazio lineare. La curva di Hilbert al primo ordine corrispondente ad un ben preciso ordinamento dei quadranti e il primo passo della sua costruzione sono mostrati in figura 3.5.

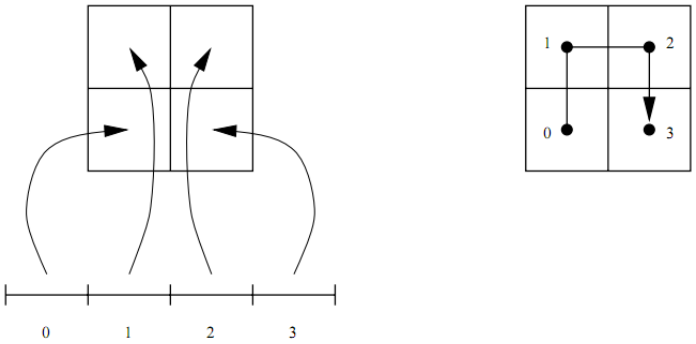


Figura 3.5 – Curva di Hilbert del primo ordine in 2 dimensioni: mapping tra i sotto-quadranti e i sotto-intervalli

Passo 2. Al secondo passo della costruzione della curva di Hilbert, il processo di divisione dei quadranti e degli intervalli è ripetuto per ciascuna delle 4 coppie di sotto-quadranti e sotto-intervalli generate alla prima iterazione. Il risultato sono 4 gruppi di 4 sotto-quadranti e 4 sotto-intervalli ciascuno di uguale dimensione e, poichè i sotto-quadranti da cui ciascun gruppo deriva sono ordinati, i gruppi stessi sono ordinati. All'interno di ciascun gruppo è stabilito un mapping tra i sotto-intervalli e i sotto-quadranti nello spazio bidimensionale ed è disegnata una curva nello stesso modo di quella descritta al primo passo. L'ordine dei sotto-quadranti all'interno di un gruppo

è scelto in maniera tale che l'ultimo sotto-quadrante di un gruppo possieda un lato in comune con il primo sotto-quadrante del gruppo successivo. Questo processo può condurre ad una curva del primo ordine con un *differente orientamento* oppure *riflessa* rispetto a quella costruita alla prima iterazione e trasforma una curva del primo ordine in una del secondo. In figura 3.6 è mostrata la curva di Hilbert del secondo ordine nella quale è stata utilizzata una rappresentazione binaria per le coordinate dei punti e per i numeri sequenza delle chiavi derivate.

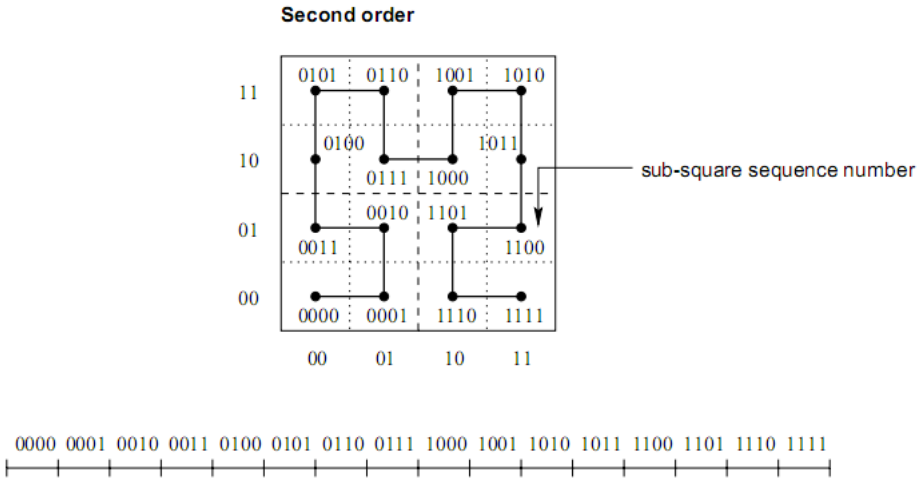


Figura 3.6 – Curva di Hilbert del secondo ordine di dimensione 2

Passo k. La curva di Hilbert di ordine k, con k>1, è concettualmente costruita sostituendo, a ciascun punto della curva di ordine (k-1), una curva di Hilbert del primo ordine opportunamente ruotata e/o riflessa in modo tale che le diverse curve possano essere connesse tra di loro e che l'ultimo punto di una curva sia adiacente al primo della curva successiva. In figura è mostrata curva di Hilbert al terzo e quarto ordine, costruite secondo il procedimento appena descritto.

Una dimostrazione della continuità del limite della curva di Hilbert è data dal fatto che i segmenti della curva uniscono sempre quadranti o sotto-quadranti che condividono un lato in comune. Il modo ricorsivo in cui è partizionato lo spazio bidimensionale (o in generale n-dimensionale) permette che punti situati all'interno di un particolare sotto-quadrante, siano vicini nell'ordinamento lineare rispetto a quanto lo siano nei confronti dei punti situati all'interno di altri sotto-quadranti della stessa dimensione.

E' possibile estendere il concetto di curva space-filling di Hilbert, presentato in precedenza per il caso bidimensionale, in generale, ad uno spazio n-dimensionale. Ad esempio, in uno spazio tridimensionale ($n = 3$), il primo passo di costruzione della curva di Hilbert prevede la divisione di un cubo di lato unitario in 8 sotto-cubi e l'ordinamento di questi ultimi in modo che sotto-cubi mappati da sotto-intervalli adiacenti possiedano una faccia in comune. Tre esempi della curva di Hilbert del primo ordine ottenuta mediante tale mapping sono riportati in figura 3.8. Si consideri l'esempo (a). Ogni sottocubo viene identificato da una stringa di 3 bits che indica, per ogni dimensione, la parte dello spazio in cui si trova il sottocubo (0 destra, 1 sinistra). I numeri in parentesi sono gli indici degli intervalli, quelli fuori dalle parentesi identificano l'ipercubo corrispondente. Come per il caso in 2 dimensioni, è possibile identificare diverse curve.

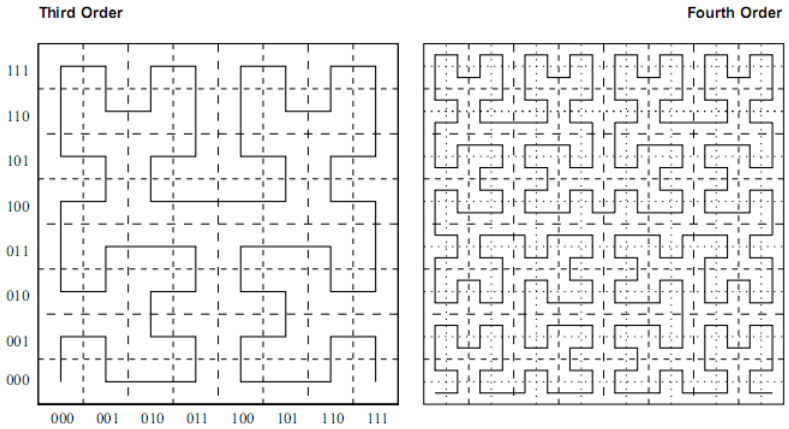


Figura 3.7 – Curva di Hilbert al terzo e quarto ordine di dimensione 2

Tramite un processo iterativo simile a quello presentato per il caso bidimensionale, dividendo i sotto-cubi e i sotto-intervalli generati all'ordine precedente si ottiene la curva di Hilbert in 3 dimensioni di ordine successivo.

Le curve di Hilbert in uno spazio bidimensionale differiscono molto rispetto a quelle di uno spazio n-dimensionale in quanto, una volta scelta la curva per il primo ordine, vi è una grande possibità di scelta su come trasformare la curva al secondo ordine. Ad esempio, in figura 3.9 sono mostrati due modi di come trasformare i primi due punti

della curva (a) del primo ordine della figura 3.8. E' anche possibile definire una curva valida del primo ordine che non può essere trasformata in una curva valida del secondo ordine.

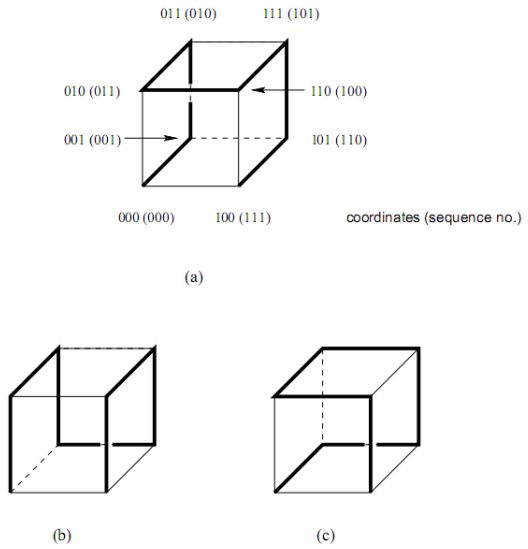


Figura 3.8 – Esempi della curva di Hilbert al primo ordine in 3 dimensioni

Queste caratteristiche complicano notevolmente la definizione degli algoritmi per la generazione della curva space-filling.

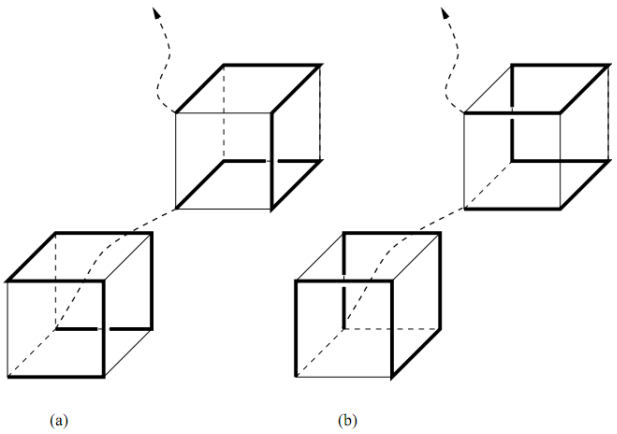


Figura 3.9 – Due esempi di trasformazione al secondo ordine (3D) dei primi due punti della figura 3.8 (a).

3.3 La curva Z-order

La *Z-order curve*, o *Morton curve* [30], fu proposta per la prima volta da *G.M. Morton* nel 1966 ed ha suscitato notevole interesse nel campo scientifico trovando applicazione in molte aree di ricerca. Una spiegazione dell'interesse che tale curva suscita nella comunità scientifica è da ricercarsi nella particolare *semplicità* con cui è possibile calcolare il *mapping* delle chiavi derivate nello spazio n-dimensionale e dal fatto che si presenta come valido *compromesso tra l'elevato grado di clusterizzazione* della curva di Hilbert e altri tipi di curve non space-filling.

La curva Z-order è, come le curve di tipo space-filling, il risultato del limite di una successione di funzioni convergenti ed è definita tramite un procedimento ricorsivo. La *Z-order curve* è definita ricorsivamente mediante trasformazioni successive di una curva f_1 definita sul segmento $[0; 1]$ con valori nel quadrato di lato unitario $Q_0 = [0; 1]^2$ nel caso bidimensionale:

$$f_1(t) : [0; 1] \rightarrow [0; 1] \times [0; 1]$$

dove t è un valore appartenente all'intervallo unitario.

Alla prima iterazione si suddivide il quadrato unitario in quattro sotto-quadrati di lato $\frac{1}{2}$, Q_{00},Q_{01},Q_{10},Q_{11}, come mostrato in figura 3.10.

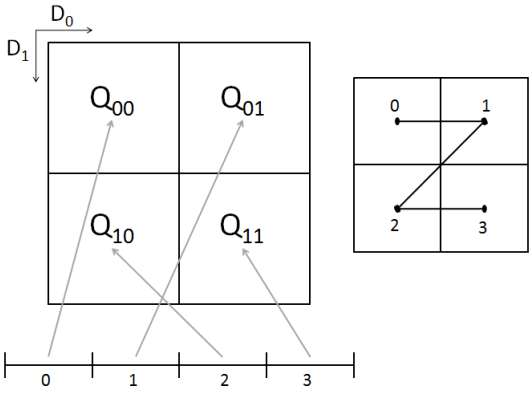


Figura 3.10 – Z-order Curve: suddivisione del quadrato unitario e mapping delle chiavi derivate

Si noti che la curva visita i quadranti in modo che l'indice di un quadrante coincida con l'indice dell'intervallo da cui tale quadrante è mappato. Questa proprietà, conservata

anche per i livelli successivi della curva, consente di semplificare notevolmente sia il processo di generazione della chiave derivata che la risoluzione delle range query, come vedremo nei capitoli 4 e 6.

La Z-order curve al secondo passo f_2, o equivalentemente del secondo ordine, si costruisce modificando la curva al passo precedente in modo che mappi $[0; \frac{1}{4}]$ in Q_{00}, $[\frac{1}{4}; \frac{1}{2}]$ in Q_{01}, $[\frac{1}{2}; \frac{3}{4}]$ in Q_{10} e $[\frac{3}{4}; 1]$ in Q_{11} e tenendo conto delle rotazioni e delle connessioni tra i vari segmenti che si vanno a creare. La curva al secondo passo f_2 sarà dunque definita come:

$$f_2(t) \in \begin{cases} Q_{00} \ t \in [0; \frac{1}{4}] \\ Q_{01} \ t \in [\frac{1}{4}; \frac{1}{2}] \\ Q_{10} \ t \in [\frac{1}{2}; \frac{3}{4}] \\ Q_{11} \ t \in [\frac{3}{4}; 1] \end{cases}$$

L'ampiezza dei quadrati al secondo passo della costruzione della Z-order curve è pari ad $\frac{1}{4}$ mentre al terzo ordine di raffinamento sarà pari ad $\frac{1}{8}$. In generale ad ogni iterazione nella costruzione della curva di livello successivo, per induzione strutturale, l'ampiezza dei sotto-intervalli si riduce di un fattore pari ad $\frac{1}{2}$ e all'ordine *k-esimo* si avrà un'ampiezza per tali intervalli pari a $\frac{1}{2^k}$.

Ad ogni passo si ripete la divisione di ciascun quadrante (o sotto-quadrante) del passo precedente in 4 quadranti più piccoli e si applicano le trasformazioni viste in precedenza. Al passo k-esimo si ottiene una funzione che mappa il segmento unitario nel quadrato unitario nel caso di due dimensioni o in un iper-cubo n-dimensionale nel caso di un numero di dimensioni maggiore.

$$f_k(t) : [0; 1] \rightarrow [0; 1]^n$$

La Z-order curve si ottiene per induzione iterando all'infinito il procedimento esposto. Alcuni esempi della curva Z-order dal primo al quarto ordine sono mostrati in figura 3.11 e mostrano chiaramente la proprietà di auto-similarità della curva per come si trasforma da un ordine al successivo.

In figura 3.11 si può notare che la curva è caratterizzata da un certo grado di discontinuità, indipendente dal numero di dimensioni nello spazio, tra ciascuna coppia di punti che non permette di classificare la curva Z-order come una curva space-filling vera e propria [22].

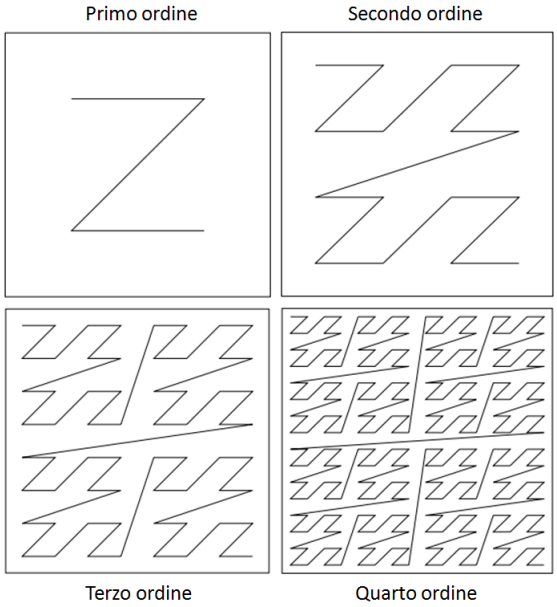


Figura 3.11 – Vari ordini di raffinamento della Z-order curve di dimensione 2

Ricordiamo che la discontinuità. riguarda il fatto che a punti vicini sulla curva possono corrispondere punti distanti nello spazio n-dimensionale. Nella Z-curve, la discontinuità è determinata dalla necessità di definire un mapping tra intervalli e quadranti che consenta di far coincidere l'indice di un intervallo con l'indice di un quadrante. In figura 3.11 la discontinuità corrisponde alle linee non orizzontali della curva.

La Z-order curve è tra le curve space-filling più utilizzate grazie alla semplicità dell'algoritmo di mapping dei punti dello spazio n-dimensionale sulle chiavi derivate nello spazio lineare e degli algoritmi di ricerca di chiavi sulla curva stessa.

3.4 La curva Gray-code

La curva *Gray-code* si basa sul concetto di *sequenza Gray-code*, la quale rappresenta una successione di numeri binari che differiscono l'uno dal precedente nel valore di un solo bit. Il procedimento per determinare una sequenza Gray-code verrà descritto nella sezione 4.1.1. Questa sequenza è originariamente attribuita a *Gray* [34] che la applicò nei suoi studi sulla trasmissione elettronica di dati e sebbene non descriva una curva space-filling vera e propria è usata spesso nel contesto di indicizzazione di dati

multidimensionali e nel progetto di una curva discontinua, la *Gray-code curve*, da parte di *Faloutsos* [33].

Sequence no.	Gray-code
0000	0000
0001	0001
0010	0011
0011	0010
0100	0110
0101	0111
0110	0101
0111	0100
1000	1100
1001	1101
1010	1111
1011	1110
1100	1010
1101	1011
1110	1001
1111	1000

Figura 3.12 – Sequenza Gray-code di lunghezza 4

Tale curva risulta essere meno discontinua rispetto alla curva Z-order. A differenza di quest'ultima infatti, nella quale si nota una certa discontinuità tra ciascuna coppia di punti, nella curva Gray-code si ha una discontinuità minore e solamente ogni sequenza di 2^n punti, pari al numero di punti su una curva del primo ordine. Inoltre, una coppia di punti sulla Z-order curve separati da una discontinuità differisce nel valore delle coordinate tra le due e le n dimensioni, mentre la stessa coppia di punti sulla curva Gray-code differisce in una sola dimensione indipendentemente dal valore di n. Il calcolo del mapping delle chiavi derivate nello spazio è più complesso rispetto a quello che si ha utilizzando la curva Z-order, ma in ogni caso, almeno concettualmente, più immediato rispetto a quello necessario per la curva di Hilbert. Un esempio della Gray-Code curve di dimensione 2 è mostrato in figura 3.13.

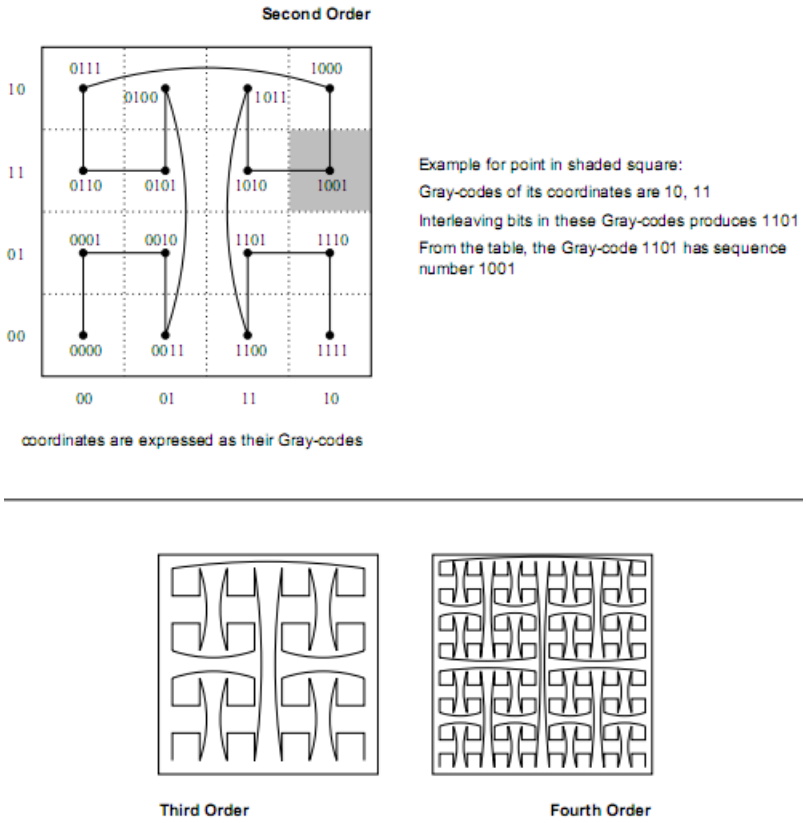


Figura 3.13 – La curva Gray-code di dimensione 2 a vari ordini di raffinamento

3.5 Le curve Scan e Snake

La *Scan curve* è caratterizzata dal modo più semplice di mappare un punto dello spazio n-dimensionale in uno spazio lineare, tramite la concatenazione delle coordinate degli n attributi, espresse come numeri interi di lunghezza fissata. Un esempio della curva Scan di dimensione 2 è mostrato in figura 3.14.

Come per la curva Z-order, in cui l'ordine dell'interleaving dei bits delle coordinate risulta flessibile e dà origine a differenti orientamenti della curva, anche in questo caso, la concatenazione delle coordinate dei punti in un singolo valore genera differenti orientamenti per la *Scan curve* che in ogni caso mantiene una certa discontinuità. Una variante della Scan curve è la *Snake curve*, mostrata anch'essa in figura 3.14, in cui coppie adiacenti di linee parallele nello spazio n-dimensionale sono collegate insieme alternativamente all'inizio o alla fine. Tale costruzione evita le discontinuità presenti

nella Scan curve. A differenza delle curve presentate in precedenza, le curve Scan e Snake possono essere definite solo per approssimazione e non tramite una divisione ricorsiva dello spazio n-dimensionale.

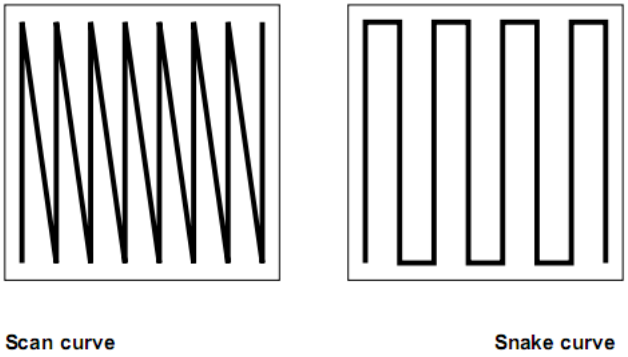


Figura 3.14 – Esempio della curva Scan e Snake di dimensione 2

Entrambe le curve Scan e Snake sono utilizzate nell'elaborazione seriale e nella trasmissione di immagini 2D, tuttavia soltanto la *Scan curve* è spesso usata come base per l'ordinamento di righe all'interno di database relazionali.

3.6 Confronto tra proprietà di clustering

In questo sezione verranno illustrate le proprietà di clustering rispetto a range query n-dimensionali di tre curve presentate: la curva di Hilbert, la curva *Gray-code* e la curva *Z-order*. Data una range query definita su uno spazio n-dimensionale come un iper-rettangolo, si definisce *cluster* un gruppo di punti appartenenti alla regione descritta dalla range query che sono consecutivi sulla curva. Come misura del clustering di una curva si utilizza il numero medio di cluster di punti all'interno del sotto-spazio definito dalla range query. In figura 3.15 è mostrato un esempio di come la stessa query bidimensionale di forma rettangolare $S_x \times S_y$ produca due cluster sulla curva *Z-order* e soltanto uno sulla curva di Hilbert.

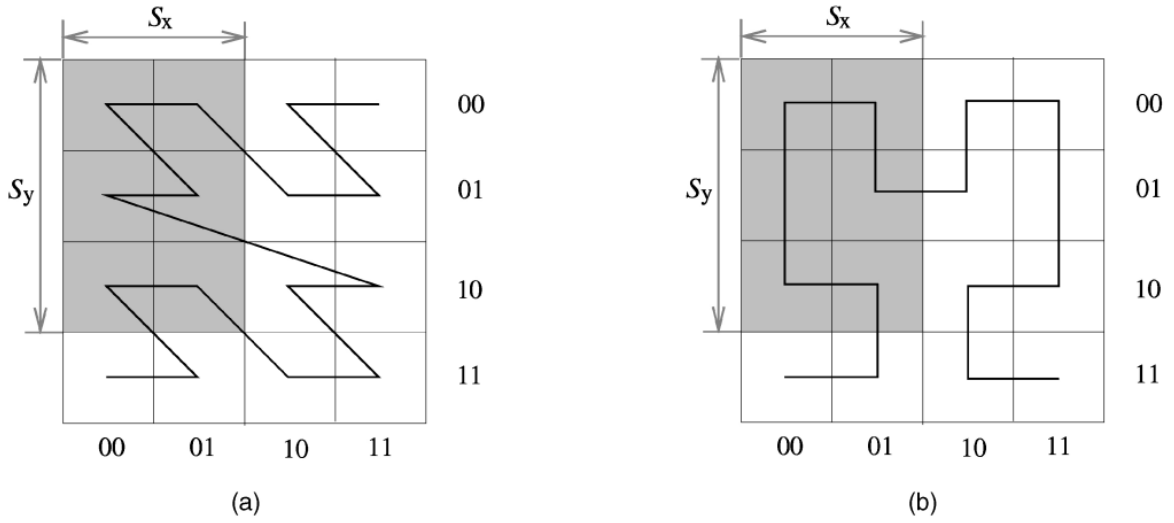


Figura 3.15 – Esempio cluster generati sulla Z-curve (a) e sulla curva di Hilbert (b) dalla stessa query $S_x \times S_y$

Un primo confronto tra le proprietà di clustering delle tre curve space-filling in oggetto, considerando solo range query di dimensione 2×2, è stata effettuato da *Jagadish* in [36]: la curva risultante che minimizza il numero di cluster è la curva di Hilbert (2), seguita dalla curva Gray-code (2.5) e dalla curva Z-order (2.625). I numeri all'interno delle parentesi successive al nome della curva indicano il numero medio di cluster generati per una range query di dimensione 2×2. In seguito, da parte di *Rong* e *Faloutsos* in [37], è stata formulata un'espressione in forma chiusa del numero medio di cluster generati da una range query sulla curva Z-order che nel caso di range query 2×2 ha dato lo stesso risultato di 2.625 cluster di media calcolato in precedenza da *Jagadish*. In generale, il numero medio di cluster sulla curva Z-order tende al valore dato dalla somma di un terzo del perimetro del rettangolo della range query con i due terzi del lato del rettangolo descritto dalla range query nella direzione sfavorita. All'inizio del 1997, da parte di *Jagadish* in [38], è stata trovata un'espressione in forma chiusa del numero medio di cluster generati anche per la curva di Hilbert in due dimensioni a partire da range query di forma quadrata 2×2 e 3×3.

Un'analisi del numero di cluster generati da una query iper-rettangolare è importante per le applicazioni che utilizzano le curve space-filling. Ad esempio, nei database in cui uno spazio multidimensionale di attributi viene mappato su uno spazio disco

unidimensionale, il numero di cluster corrisponde al numero di accessi non consecutivi al disco per il reperimento dei dati.

Nel caso di *HASP*, il numero di cluster, influenza il numero di confronti che devono essere effettuati tra i cluster generati appartenenti alla range query e le aggregazioni di chiavi derivate contenute nei nodi dell'albero *HASP*.

Sia N_n il numero medio di cluster dentro un iper-rettangolo di n-dimensioni, il calcolo del numero medio di cluster formati da una range query rettangolare sulla curva di Hilbert di dimensione n e ordine k , denotata come H_k^n, è formulato in [35] ed è espresso dal seguente teorema:

Teorema 3.6.1. *In uno spazio n-dimensionale sufficientemente ampio mappato mediante la curva di Hilbert di dimensione n e ordine k, H_k^n, sia S_q l'area totale della superficie di una query q, allora vale:*

$$\lim_{k\to\infty} N_n = \frac{S_q}{2n}$$

La dimostrazione del teorema può essere trovata in [35].

Grazie al precedente teorema è possibile affermare che la curva di Hilbert possiede un grado di clusterizzazione migliore rispetto alla curva Z-order. Nel caso bidimensionale infatti, il numero medio di cluster per la curva di Hilbert è approssimativamente pari ad un quarto del perimetro del rettangolo descritto dalla range query, mentre per la Z-order curve è pari circa alla somma di un terzo del perimetro del rettangolo della range query con i due terzi del lato del rettangolo nella direzione sfavorita.

Un corollario al teorema precedente è il seguente:

Corollario 3.6.2. In uno spazio n-dimensionale sufficientemente ampio mappato sulla curva di Hilbert di dimensione n e ordine k, H_k^n, sono soddisfatte le seguenti due proprietà:

- dato un iper-rettangolo $s_1 \times ... \times s_n$, il numero medio di cluster è pari a:

$$\lim_{k\to\infty} N_q = \frac{1}{n}\sum_{i=1}^{n}(\frac{1}{s_i}\prod_{j=1}^{n} s_j)$$

- dato un iper-cubo di lato s, il numero di cluster è pari a:

$$\lim_{k\to\infty} N_n = s^{n-1}$$

Il teorema precedente indica che se la dimensione della griglia cresce all'infinito, il numero medio di cluster tende a diventare metà dell'area di una certa query diviso per il numero di dimensioni dello spazio. Non indica però quanto rapidamente si abbia la convergenza verso questo valore, valido solo per il risultato medio e non indica niente circa la varianza dei risultati rispetto al caso medio. Per questa ragione è interessante riportare anche i risultati sperimentali ottenuti in [35]. Nelle figure 3.16 e 3.17 sono mostrati i risultati sperimentali ottenuti da una simulazione esaustiva eseguita su uno spazio bidimensionale di grandezza $1K \times 1K$ e da una simulazione statistica su uno spazio tridimensionale $32K \times 32K \times 32K$ per ciascuna delle tre curve space-filling in esame al fine di determinare quale curva generi un minor numero di cluster sia nel *caso medio* che nel *caso pessimo* [35]. Nelle figure 3.16(a) e (b) vengono riportati i risultati per query rispettivamente di forma quadrata e circolare in uno spazio bidimensionale. Nelle figure 3.16(c) e (d) vengono riportati gli stessi risultati per uno spazio tridimensionale. Si riportano rispettivamente i risultati al variare della lunghezza del lato del quadrato e del raggio del cerchio.

Si noti come nel grafico in figura 3.16(a), i risultati per la curva Z-order e per la curva Gray-code sono esattamente uguali. La curva di Hilbert ottiene un grado di clusterizzazione decisamente migliore rispetto alle altre due curve sia nel caso medio che nel caso pessimo. Ad esempio, per una query di forma quadrata nello spazio bidimensionale, la curva di Hilbert genera un numero nettamente inferiore di cluster, con un miglioramento rispetto alle altre due curve fino a circa il 48% nel caso medio e al 43% nel caso peggiore. Nel caso di una query sferica, quindi in uno spazio tridimensionale, la curva di Hilbert è caratterizzata da una riduzione del numero di cluster nel caso peggiore rispetto alla curva Z-order fino al 28% e rispetto alla curva Gray-code fino al 18%, mentre nel caso medio, rispetto alla curva Z-order, la curva di Hilbert genera un numero di cluster inferiore fino al 31%, mentre rispetto alla curva Gray-code fino al 22% meno. E' importante notare come non sempre la curva Gray-code generi un numero di cluster minore rispetto alla curva Z-order e come ciò è in contrasto con lo studio precedente [36], nel quale la curva Gray-code otteneva una miglior grado di clustering rispetto alla curva Z-order nel caso di query di forma quadrata 2×2. In particolare per query di forma circolare bidimensionali, figura 3.16(b) e 3.17(b), la curva Gray-code si comporta in maniera peggiore rispetto alla curva Z-order sia nel caso medio che nel caso peggiore. Al contrario, per query di forma quadrata in uno spazio bidimensionale, la curva Gray-code si rivela migliore nel numero di cluster nel

caso medio rispetto alla curva Z-order per una quantità trascurabile (figura 3.17(a)), mentre le due curve ottengono la stessa misura di *clustering* nel caso pessimo (figura 3.16(a)). In uno spazio tridimensionale invece, la curva Gray-code si rivela decisamente migliore rispetto alla curva Z-order per entrambi i tipi di query sia nel caso medio che nel caso peggiore.

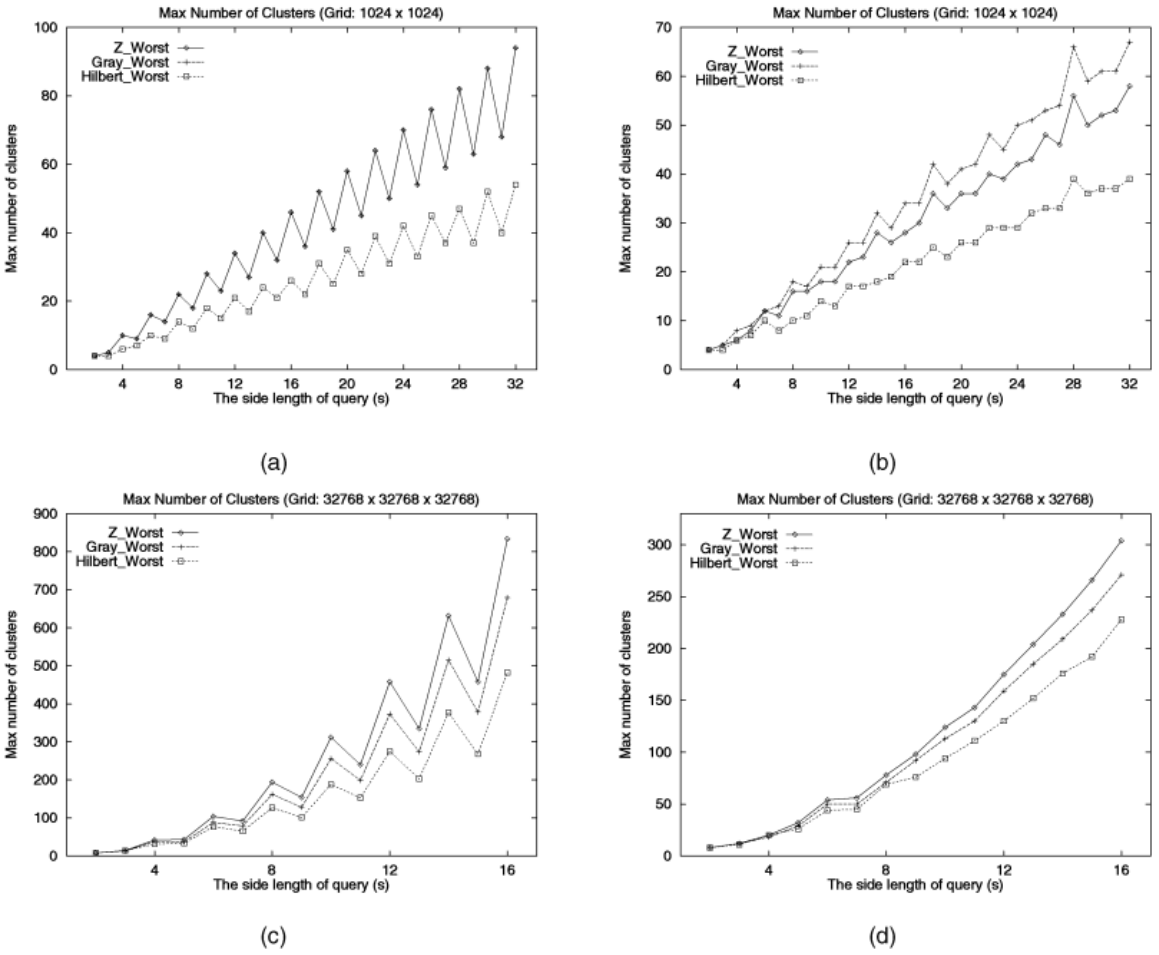


Figura 3.16 – Numero di cluster nel caso peggiore per tre differenti curve space-filling

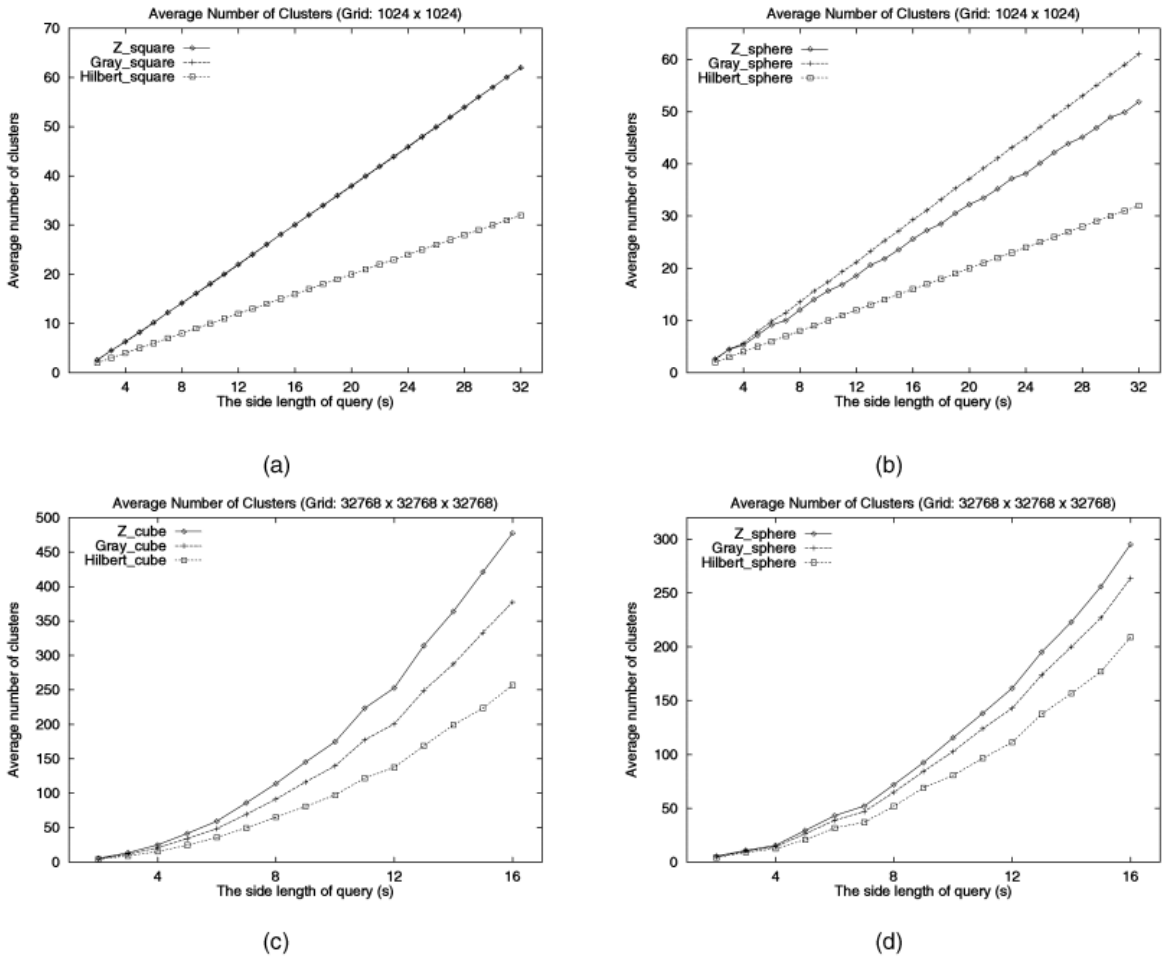


Figura 3.17 – Numero medio di cluster per tre differenti curve space-filling

3.7 Conclusioni

Dall'analisi delle proprietà di clustering delle curve space-filling riportata nella sezione precedente 3.6, si evince come la curva di Hilbert possieda un miglior grado di clusterizzazione rispetto alle altre due curve, Z-order e Gray-code, sia nel caso medio che nel caso pessimo. La definizione di un algoritmo per la generazione delle chiavi derivate di Hilbert e la risoluzione di range query su questa curva, tuttavia, è molto complessa. La curva Z-order, sebbene possieda un grado di clusterizzazione delle chiavi minore, è caratterizzata da algoritmi molto semplici per la generazione delle chiave derivate e di

intersezione con le range query. La curva Gray-code possiede caratteristiche riguardo al grado di clusterizzazione e alla semplicità degli algoritmi paragonabili a quelli per la curva Z-order. Per la scelta della curva da adottare in *HASP* si è scelta una soluzione di compromesso, in quanto sebbene la scelta sia ricaduta sulla curva Z-order per la maggiore semplicità degli algoritmi, si cerca di diminuire il numero di cluster da esaminare tramite una soluzione *guidata dai dati* che verrà presentata nei capitoli 5 e 6.

Capitolo 4

Generazione della chiave derivata

In questo capitolo saranno presentati e successivamente confrontati gli algoritmi per il calcolo delle chiavi derivate su due diverse curve di tipo space filling: *la curva di Hilbert e la Z-Curve.* Entrambi gli algoritmi saranno seguiti da un esempio di generazione di chiave derivata sulla curva space filling in oggetto, al fine di rendere più semplice la loro comprensione.

4.1 Curva di Hilbert

L'algoritmo per la generazione della chiave derivata è quello proposto in [22]. La generazione di una chiave derivata, appartenente alla curva space filling di Hilbert, si compone di due parti principali: una prima parte riguardante la creazione di una tabella di generazione del diagramma degli stati e una seconda parte riguardante la costruzione della chiave derivata vera e propria a partire da questa tabella.

4.1.1 Algoritmo per la creazione della tabella di generazione del diagramma degli stati

Prima di descrivere l'algoritmo per la creazione della tabella di generazione del diagramma degli stati è necessario definire il concetto di *n-point.*

Definizione 4.1.1. *Un n-point è l'insieme delle coordinate di un bit di un punto appartenente alla curva del primo ordine, concatenati in un unico valore di n bits. Questi punti rappresentano i punti centrali dei sotto-quadranti in cui lo spazio è partizionato.*

Il primo passo per la generazione della chiave derivata di Hilbert prevede la costruzione della tabella di generazione del diagramma degli stati o *generator table*. La funzione principale di questa tabella è quella di descrivere come una particolare curva del primo ordine sia trasformata in una del secondo ordine. Una generator table si compone di un numero ben preciso di righe e colonne; ciascuna riga corrisponde ad un punto che giace su una curva del primo ordine, quindi sono presenti in totale r^n righe, nel nostro caso r=*2* e n pari al numero di dimensioni della curva. Le righe sono ordinate in base al valore dei numeri di sequenza (*sequence numbers*) dei punti sulla curva e insieme definiscono la curva del primo ordine che corrisponde al primo stato nel diagramma degli stati. Questa tabella contiene al suo interno l'informazione necessaria per eseguire la trasformazione di una curva del primo ordine in una del secondo, in particolare indica come sostituire ciascun punto appartenente alla curva del primo ordine con un'altra curva del primo ordine eventualmente riflessa oppure ruotata. Queste curve, con cui sono sostituiti i punti, saranno chiamate, nel contesto del diagramma degli stati, stati successivi o *next-states*. Le colonne che compongono la generator table sono le seguenti: Y, X_1, X_2, δY e *T(Y)*.

Y	X_1	X_2	δY	T(Y)	
00	00	00	01	0	1
		01		1	0
01	01	00	10	1	0
		10		0	1
10	11	00	10	1	0
		10		0	1
11	10	11	01	0	-1
		10		-1	0

Figura 4.1 – *Generator table* nel caso bidimensionale

In figura 4.1 è mostrato un esempio di generator table. La colonna Y contiene tutti i membri del dominio del mapping dei punti da uno spazio unidimensionale ad uno n-dimensionale per una curva del primo ordine. Ciascun valore esprime la distanza dall'inizio della curva e rappresenta la chiave derivata di un punto sulla curva. In ogni riga è contenuto un valore numerico distinto compreso nell'intervallo $[0, ..., r^n - 1]$, composto da un numero di cifre pari al numero di dimensioni n e di cui ciascuna cifra può assumere un valore compreso nell'intervallo $[0, ..., r - 1]$. I valori presenti

nella colonna Y della tabella di generazione del diagramma degli stati sono ordinati in ordine crescente. Per esempio nel caso bidimensionale, quindi con n pari a due, la colonna Y della generator table conterrà i seguenti valori: 00_2, 01_2, 10_2 e 11_2.

I valori presenti nella colonna X_1 della tabella esprimono l'ordinamento dei punti sulla curva del primo ordine. Il metodo utilizzato per calcolarli [22], si basa sull'utilizzo della sequenza *Gray-code*. La *Gray-code sequence* è una sequenza di numeri binari in cui qualsiasi coppia di membri consecutivi differiscono tra di loro per il valore di un bit in un'unica posizione.

L'algoritmo per il calcolo della sequenza Gray-code di ordine n è il seguente:

1. Si inizializza la sequenza come [0, 1].

2. La sequenza viene raddoppiata, aggiungendo i membri della sequenza al passo precedente in ordine inverso.

3. A ciascun membro della prima metà della sequenza viene aggiunto un bit di valore 0 come bit più significativo, mentre ai membri della seconda metà un bit di valore 1.

4. I passi 2 e 3 vengono ripetuti per produrre sequenze successive sempre più lunghe, raddoppiando la lunghezza ad ogni iterazione. Il numero di bit, necessario per rappresentare ciascun elemento della sequenza, aumenta di una unità ad ogni iterazione.

5. Dopo n iterazioni la sequenza contiene 2^n elementi distinti ciascuno composto da n bits. Tale sequenza viene chiamata *Gray-code sequence*.

Esempio 4.1.2. Mostriamo di seguito un esempio di generazione della *Gray-code sequence*.

[0, 1]
[0, 1, 1, 0]
[00, 01, 11, 10] = *Gray-code sequence* di ordine 2
[00, 01, 11, 10, 10, 11, 01, 00]
[000, 001, 011, 010, 110, 111, 101, 100] = *Gray-code sequence* di ordine 3
...

La colonna X_1 della generator table contiene i membri della sequenza Gray-code così calcolati. Le colonna Y contiene le chiavi derivate dei punti sulla curva del primo ordine, mentre la colonna X_1 contiene le coordinate di quei punti. La terza colonna della generator table è la colonna X_2, in cui in ciascuna entry è contenuta una coppia di valori ognuno di n bits. Ciascuna coppia di valori corrisponde alle prime ed ultime coordinate dei punti su una curva del primo ordine, nei quali un punto preso dalla colonna X_1 viene mappato nella trasformazione al secondo ordine.

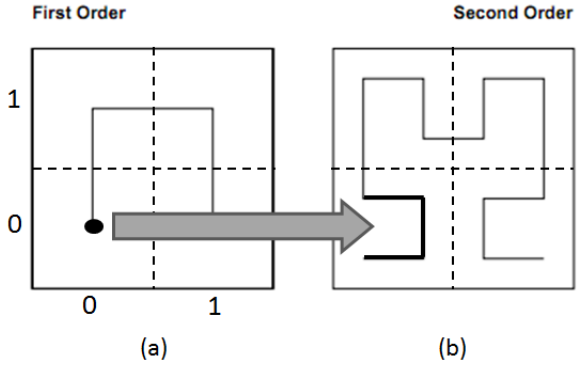


Figura 4.2 – Mapping della curva del primo ordine nella curva di Hilbert del secondo ordine

Consideriamo ad esempio la figura 4.2(a) ed in particolare il punto di coordinate 00 a cui corrisponde la chiave derivata 0 sulla curva del primo ordine. Il punto 00 viene trasformato in una curva del primo ordine ruotata rispetto a quella originale, come mostrato in figura 4.2(b). Le coordinate del primo punto della curva ruotata sono 00, mentre quelle dell'ultimo punto sono 01 che sono proprio i valori riportati nella colonna X_2.

Le entrate di questa colonna, per una curva di n dimensioni, sono popolate tramite i membri della *X2Gray-code sequence* di ordine n.

L'algoritmo per il calcolo della sequenza *X2Gray-code* di ordine n è il seguente:

1. Si inizializza la sequenza di ordine 1 come [0, 1, 0, 1].
2. Si inizializza la sequenza di ordine n pari alla sequenza di ordine *n-1*
3. Si sostituisce il valore dell'ultimo membro della sequenza con il valore del penultimo membro, così da ottenere gli ultimi due membri di valore uguale.

4. A ciascun membro della sequenza, tranne che all'ultimo, si aggiunge come prefisso un bit pari a 0, mentre all"ultimo membro un bit pari ad 1.

5. Si raddoppia la lunghezza della sequenza fin qui ottenuta in maniera speculare, in modo tale che l'ultimo membro sia uguale al primo, il penultimo uguale al secondo e così via.

6. Si invertono i valori di tutti i bit più significativi dei membri della seconda metà della sequenza.

7. Si ripetono i punti dal numero 2 al numero 6 per ottenere la *X2Gray-code sequence* di ordine n.

Esempio 4.1.3. Mostriamo di seguito un esempio di generazione della *X2Gray-code sequence*.

[0, 1, 0, 1]
[0, 1, 0, 0]
[00, 01, 00, 10]
[00, 01, 00, 10, 10, 00, 01, 00]
[00, 01, 00, 10, 00, 10, 11, 10] = *X2Gray-code sequence* di ordine 2
...

I valori contenuti nella colonna δY, per la curva di Hilbert, sono composti da n bits ciascuno e di questi uno soltanto possiede valore pari ad uno, mentre tutte le altre *n-1* cifre hanno valore pari a zero. La posizione del bit settato ad uno indica l'asse su cui giacciono il primo e l'ultimo punto della curva del primo ordine, nella quale il punto nella colonna X_1 viene mappato nella trasformazione al secondo ordine. In altri termini δY indica quale è la dimensione su cui vi è variazione tra le coordinate dei due punti che indicano l'inizio e la fine della curva. Ciascuna riga contiene un numero, il cui valore è determinato dalla *differenza bit a bit* dei due valori presenti nella colonna X_2 della medesima riga. Ad esempio nel caso bidimensionale, nel caso in cui i valori della colonna X_2 siano rispettivamente 11 e 10, il valore presente alla colonna δY della stessa riga sarà 01.

La colonna *T(Y)* contiene una matrice di trasformazione di dimensione $n \times n$ per ogni entry, la quale esprime come una curva del primo ordine differisca dalla curva del

primo ordine definita dai valori contenuti dalle colonne Y e X_1. In particolare tale matrice indica come una curva del primo ordine, le cui coordinate del primo ed ultimo punto, espresse come *n-points*, e contenute nella coppia di valori della colonna X_2, differisca dallo stato iniziale S_0 e quindi implicitamente incapsula lo stato successivo o *next-state* per ciascun punto (contenuto nella colonna X_1) nello stato S_0. La matrice di trasformazione per lo stato S_0 stesso è uguale alla matrice identità.

L'algoritmo per il calcolo della matrice di trasformazione di ciascuna riga è il seguente:

1. Si setta la prima riga della matrice pari al valore contenuto nella colonna δY della medesima entry a cui la matrice di trasformazione si riferisce.

2. Per ciascuna riga della matrice dalla seconda alla n-esima, si pone la riga i-esima pari al valore della riga (i-1)-esima, a cui viene applicato uno shift circolare verso destra di una posizione.

3. Si aggiustano i segni degli elementi della matrice con valore diverso in base alla regola di Bially definita in [23]: se l'i-esimo bit del primo numero della coppia contenuta nella colonna X_2 della medesima riga è diverso da zero allora l'elemento non zero della i-esima colonna della matrice è posto uguale a -1.

4.1.2 Algoritmo per la generazione della chiave derivata di Hilbert

L'algoritmo utilizzato per la generazione della chiave derivata a partire da n attributi numerici è stato proposto da Jonathan Lawder in [22]. Lo scopo dell'algoritmo è quello di determinare un'associazione univoca tra il valore di n attributi ed un punto appartenente alla curva di Hilbert, le cui coordinate compongono la chiave derivata o *Hilbert derived-key*. Si effettua dunque un passaggio da un sistema n-dimensionale formato dagli n attributi ad un sistema lineare formato dalle chiavi derivate di Hilbert.

Il primo passo dell'algoritmo di generazione della chiave derivata di Hilbert prevede la creazione della *generator table* di dimensione n necessaria per le fasi successive ed il suo ordinamento rispetto ai valori della colonna X_1. Come descritto in seguito, l'algoritmo effettua diversi accessi alla generator table in base ai valori contenuti nella colonna X_1. A questo scopo si effettua l'ordinamento della generator table in base

ai valori della colonna X_1, in modo da poter effettuare un *accesso diretto* alla tabella in base al valore di X_1. Le strutture dati utilizzate nel corso dell'algoritmo sono una matrice di dimensione $n \times n$, $matr_{Sc}$, ad indicare la matrice di trasformazione dello stato corrente ed un array di n posizioni, *signs,* utilizzato per mantenere i segni degli elementi della matrice diversi da zero negativi. La posizione i-esima di questo array avrà valore pari a uno, se nella i-esima colonna della matrice l'elemento non nullo ha valore pari a -1, mentre avrà valore pari a zero nel caso in cui tale elemento abbia valore pari ad uno. La matrice di trasformazione dello stato corrente viene modificata ad ogni iterazione del ciclo principale dell'algoritmo ed identificherà lo stato corrente in cui l'algoritmo si trova; ad ogni aggiornamento di questa matrice corrisponde un aggiornamento dell'*array* dei segni degli elementi.

Un esempio della matrice di trasformazione dello stato corrente, $matr_{Sc}$, e della struttura dati che ne contiene i segni, s*igns*, è rappresentato in figura 4.3:

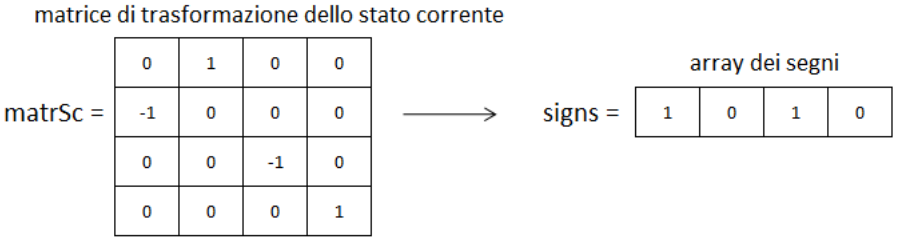


Figura 4.3 – Esempio di matrice di trasformazione dello stato corrente

Il secondo passo dell'algoritmo prevede la creazione di una variable P il cui valore è dato dalla concatenazione delle rappresentazioni binarie dei valori degli n attributi di input. La notazione a_j^i sta ad indicare il bit j-esimo dell'attributo i-esimo.

$$P = \{a_0^0\ a_1^0\ ...\ a_{k-1}^0\ a_0^1\ a_1^1\ ...\ a_{k-1}^1\ ...\ a_0^{n-1}\ a_1^{n-1}\ ...\ a_{k-1}^{n-1}\ \}$$

La variable P avrà dunque una dimensione pari a $n \times k$ bits, dove k esprime il numero di bits necessari per la rappresentazione binaria dell'attributo con valore maggiore e n il numero di attributi (dimensioni). La variabile k esprime più precisamente l'ordine di raffinamento della curva ovvero l'intervallo di valori $[0\,;2^k-1]$, in cui il valore di ciascun attributo è contenuto. La codifica binaria dei valori degli altri n attributi

in input verrà espressa utilizzando k bits per ciascuna, eventualmente aggiungendo uno o più bit con valore pari a zero nelle posizioni più significative. Un esempio del valore assunto dalla variabile P, con $n = 4$ e $k = 5$, è riportato nella figura 4.4.

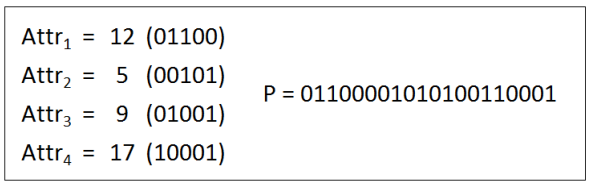


Figura 4.4 – Esempio di costruzione della variabile P

La matrice dello stato corrente all'inizio dell'algoritmo è inizializzata con la matrice identità, mentre l'array che rappresenta i segni degli elementi non nulli della matrice contiene il valore zero in tutte le posizioni. Le altre inizializzazioni, effettuate all'inizio dell'algoritmo, riguardano la variabile HK che conterrà la chiave derivata di Hilbert e una variabile intera i utilizzata per il controllo della terminazione del ciclo principale. La variabile i inizialmente avrà un valore pari ad uno, mentre HK è inizializzata con il valore zero.

La parte principale dell'algoritmo è composta da un ciclo che viene ripetuto per k iterazioni, con k che esprime l'ordine della curva di Hilbert ed è mostrata in figura 4.5.

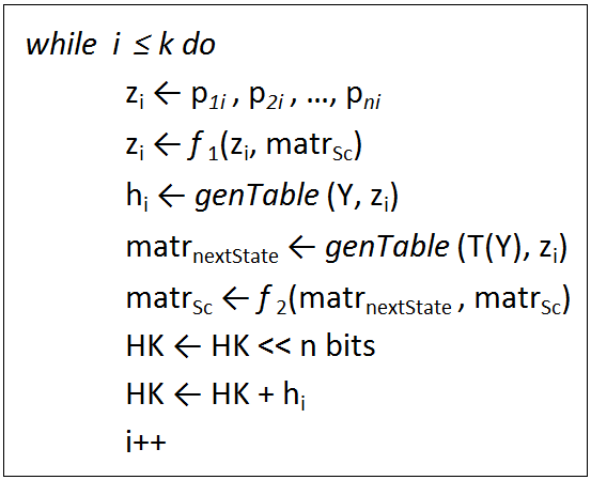


Figura 4.5 – Algoritmo per la generazione della chiave derivata di Hilbert

La prima istruzione del ciclo assegna alla variabile z_i un valore pari ai bit i-esimi di ciascun attributo che forma la variabile P, quindi z_i sarà composta ad ogni iterazione da n bits. Il valore di z_i corrisponde ad un *n-point*.

Nella seconda istruzione la variabile z_i è aggiornata con il valore restituito dal calcolo della funzione f_1 avente come parametri lo stesso valore corrente assunto da z_i e la matrice di trasformazione dello stato corrente. La funzione f_1 è composta di due step successivi: il primo prevede il calcolo dell'or esclusivo (XOR) tra il valore corrente della variabile z_i e l'array *signs* che contiene i segni della matrice di trasformazione dello stato corrente, mentre nel secondo si effettua una permutazione dei bit del valore ottenuto come risultato del primo passo a seconda del valore degli elementi della matrice $matr_{Sc}$ corrente. Per ciascuna riga m della matrice, se esiste un valore non nullo nella colonna j-esima e nella posizione j-esima del risultato dell'or esclusivo vi è un bit diverso da zero, allora si pone il bit nella posizione m-esima pari al valore di quello presente nella posizione j-esima e il bit nella posizione j-esima a zero. Nel caso in cui il bit in posizione j-esima sia già stato modificato in un'iterazione precedente, il suo valore rimane immutato. Il confronto tra gli elementi della matrice di trasformazione dello stato corrente ed il valore dei bit del risultato dell'or esclusivo avviene sempre sul valore ottenuto dal calcolo dopo il primo passo della funzione f_1, senza tener conto delle modifiche che possono essere state effettuate sui bit nelle iterazioni precedenti del secondo step.

Esempio 4.1.4. Un esempio del valore di z_i ottenuto dal calcolo della funzione f_1 avente come parametri la matrice di trasformazione dello stato corrente in figura 4.3 ed un valore di $z_i = 0110$, è il seguente:

Primo passo:
z_i =signs $\bigoplus z_i$
$z_i = 1010 \bigoplus 0110 \ = 1100$

Secondo passo:
$m = 0,\ j = 1 : \text{matr}_{\text{Sc}}(0,0) \neq 0 \ \wedge z_i[1] \neq 0 \rightarrow z_i = 1000$
$m = 1,\ j = 0 : \text{matr}_{\text{Sc}}(1,0) \neq 0 \ \wedge z_i[0] \neq 0 \rightarrow z_i = 1100$
$m = 2,\ j = 2 : \text{matr}_{\text{Sc}}(2,2) \neq 0 \ \wedge z_i[2] = 0$ nessuna trasformazione
$m = 3,\ j = 3 : \text{matr}_{\text{Sc}}(3,3) \neq 0 \ \wedge z_i[3] = 0$ nessuna trasformazione

Valore finale di $z_i = 1100$

Il valore di z_i calcolato serve per accedere alla generator table e ricavare una parte della chiave derivata.

Nella terza istruzione si memorizza nella variabile h_i il valore della k-esima parte della chiave derivata di Hilbert, ricavato dal valore contenuto nella colonna Y della riga z_i-esima della generator table creata inizialmente.

Sempre mediante la generator table si ricava nella quarta istruzione, dalla medesima riga precedente indirizzata dalla variabile z_i, la matrice di trasformazione per lo stato successivo che coincide con la matrice di trasformazione contenuta nella colonna *T(Y)*. L'aggiornamento della matrice dello stato corrente avviene nella quinta istruzione del codice tramite l'esecuzione di una funzione f_2, avente come parametri la matrice di trasformazione dello stato corrente e la matrice di trasformazione per lo stato successivo ricavata dalla generator table. Il risultato restituito dal calcolo della funzione f_2 con questi parametri costituisce la nuova matrice di trasformazione dello stato corrente. La funzione f_2, come la funzione f_1 è definita mediante due step successivi.

Il primo passo prevede il calcolo della nuova matrice di trasformazione per lo stato corrente (matrice A), come permutazione delle righe della matrice di trasformazione per lo stato successivo ricavata dalla generator table al passo precedente (matrice C - matrice di trasformazione per z_i nello stato S_0) in base al valore degli elementi della matrice di trasformazione dello stato corrente attuale (matrice B). La matrice A viene calcolata in base alle seguente regola: *se nella riga i-esima della matrice B esiste un bit non nullo nella colonna j-esima, allora la la riga i-esima della matrice A assume il valore degli elementi della riga j-esima della matrice C.*

Il secondo step della funzione f_2 modifica i segni degli elementi della matrice A a seconda del valore degli elementi delle matrici B e C in maniera differente a seconda dei seguenti casi:

- se il segno dell'elemento non nullo nella riga m-esima della matrice B è negativo e il segno dell'elemento non nullo nella riga j-esima della matrice C è positivo, allora il segno dell'elemento diverso da zero nella riga m-esima della matrice A diventa negativo.

- se il segno dell'elemento non nullo nella riga m-esima della matrice B è positivo e il segno dell'elemento non nullo nella riga j-esima della matrice C è negativo, allora il segno dell'elemento diverso da zero nella riga m-esima della matrice A rimane immutato.

- se i segni dell'elemento non nullo nella riga m-esima della matrice B e di quello non nullo nella riga j-esima della matrice C sono entrambi positivi, allora il segno dell'elemento diverso da zero nella riga m-esima della matrice A rimane immutato.

- se i segni dell'elemento non nullo nella riga m-esima della matrice B e di quello non nullo nella riga j-esima della matrice C sono entrambi negativi, allora il segno dell'elemento diverso da zero nella riga m-esima della matrice A diventa positivo.

Una volta terminato il calcolo della funzione f_2 ed ottenuta la matrice di trasformazione per il prossimo stato, si aggiorna la matrice di trasformazione dello stato corrente con questa e l'*array signs* con i nuovi valori.

Nella sesta e settima istruzione del corpo del ciclo dell'algoritmo si effettua rispettivamente uno shift verso sinistra di n bits del valore della variabile *HK,* che contiene la chiave derivata di Hilbert già calcolata e si somma a questa la k-esima parte della chiave, calcolata nella terza istruzione. Al termine dell'algoritmo la variabile *HK* conterrà la chiave derivata di Hilbert, di dimensione $n \times k$ bits, generata a partire dal valore degli n attributi di input.

4.1.3 Esempio di generazione chiave derivata di Hilbert

Di seguito è riportato un esempio completo della generazione della chiave derivata di Hilbert a partire dal valore di tre attributi ($n = 3$) con valori compresi nell'intervallo $[0\,;16]$, quindi con un ordine della curva di Hilbert pari a $k = 4$. Supponiamo di voler generare la chiave derivata di Hilbert dei tre attributi con i seguenti valori:

$$\begin{aligned} \text{Attr}_1 &= 12\ (1100_2) \\ \text{Attr}_2 &= 9\ (1001_2) \\ \text{Attr}_3 &= 5\ (0101_2) \end{aligned}$$

la variabile P assumerà il valore $P = 110010010101$, inizialmente la matrice di trasformazione dello stato corrente sarà pari alla matrice identità e l'array contenente i segni della matrice avrà tutti gli elementi pari a zero. Nella prima iterazione del ciclo la variabile i ha valore pari ad uno e ciò implica un valore della variabile $z_1 = 110$ dato dalla concatenazione del primo bit di ciascun attributo. Il valore della variabile z_1 in seguito al primo e al secondo step della funzione f_1 rimane invariato, in quanto signs $= 000$ e quindi z_1XOR signs non modifica il valore di z_1, mentre nel secondo step non vi è alcuno scambio di posizione tra i bit che compongono z_1. Il valore della variabile h_1 viene ricavato dal valore contenuto nella colonna Y della generator table alla riga z_i-esima. Poichè il valore di $z_1 = 110$, si prende la sesta entry della tabella, ordinata all'inizio dell'algortimo secondo i valori della colonna X_1, ottenendo un valore per la variabile $h_1 = 100$ che rappresenta il primo segmento della chiave derivata. In figura 4.6 è mostrata la *generator table* nel caso in cui $n = 3$ ordinata secondo i valori della colonna Y. Ricordiamo che le righe della tabella sono numerate a partire da 0 fino a $2^n - 1$.

Dalla medesima riga, ma dalla colonna *T(Y)*, si ricava il valore della matrice di trasformazione per il prossimo stato:

$\text{matr}_{\text{nextState}} =$

1	0	0
0	-1	0
0	0	-1

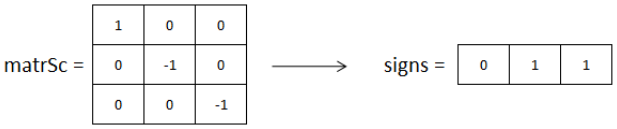
Il calcolo della nuova matrice di trasformazione dello stato corrente (matrice A) prevede la creazione della nuova matrice come permutazione delle righe della matrice di trasformazione del prossimo stato appena calcolata (matrice C) in base ai valori della matrice di trasformazione dello stato corrente attuale (matrice B). La matrice di trasformazione dello stato corrente attuale è la matrice identità. Poichè nelle righe m-esime della matrice B esiste un bit non nullo nella colonna j-esima solo nei casi in cui $m = j$, allora la matrice A dopo il primo step della funzione f_2 diventa uguale alla matrice C. Nel secondo step di f_2 si applica la regola in cui i segni degli elementi

sono entrambi positivi per *(m*=0, *j*=0) e la regola *se il segno dell'elemento non nullo nella riga m-esima della matrice B è positivo e il segno dell'elemento non nullo nella riga j-esima della matrice C è negativo* sia per *(m*=1, *j*=1) che per *(m*=2, *j*=2). Tali regole mantengono invariati i segni e quindi la matrice A rimane la stessa determinata al primo step.

Y	X_1	X_2	δY	$T(Y)$
000	000	000	001	0 0 1
		001		1 0 0
				0 1 0
001	001	000	010	0 1 0
		010		0 0 1
				1 0 0
010	011	000	010	0 1 0
		010		0 0 1
				1 0 0
011	010	011	100	1 0 0
		111		0 -1 0
				0 0 -1
100	110	011	100	1 0 0
		111		0 -1 0
				0 0 -1
101	111	110	010	0 -1 0
		100		0 0 1
				-1 0 0
110	101	110	010	0 -1 0
		100		0 0 1
				-1 0 0
111	100	101	001	0 0 -1
		100		-1 0 0
				0 1 0

Figura 4.6 – *Generator table* per il caso tridimensionale

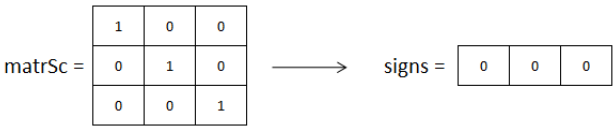
Di seguito è riportata la nuova matrice di trasformazione dello stato corrente dopo l'applicazione di entrambi gli step della funzione f_2:



Al termine della prima iterazione HK = 100_2.

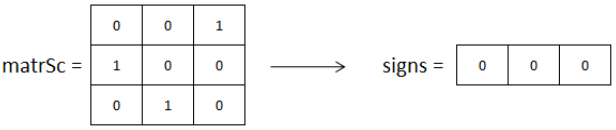
Nella seconda iterazione inizialmente $z_2 = 101$ e dopo il calcolo del nuovo valore di z_2 tramite l'applicazione della funzione f_1, la variabile h_2 assume nuovamente il

valore $h_2 = 100$. La nuova matrice di trasformazione dello stato corrente, in seguito all'applicazione della funzione f_2 , è:



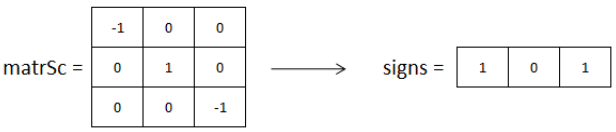
Al valore corrente della variabile HK viene applicato uno shift verso sinistra di 3 posizioni diventando così pari a 100000 e, successivamente, aggiunta la 2^a parte della chiave di Hilbert calcolata (h_2). Al termine della seconda iterazione il valore della variabile HK è pari a HK = 100100_2.

Nella terza iterazione inizialmente $z_3 = 000$ e dopo il calcolo del nuovo valore di z_3 tramite l'applicazione della funzione f_1, la variabile h_3 assume il valore $h_3 = 000$. La nuova matrice di trasformazione dello stato corrente dopo l'applicazione della funzione f_2 è:



Al valore corrente della variabile HK viene applicato uno shift verso sinistra di 3 posizioni, diventando così pari a 100100000 e successivamente aggiunta la 3^a parte della chiave di Hilbert calcolata (h_3). Il valore della variabile HK al termine della terza iterazione è: HK = 100100000_2.

All'inizio della quarta ed ultima iterazione si ha $z_4 = 011$ e dopo il calcolo del nuovo valore di z_4 tramite l'applicazione della funzione f_1, la variabile h_4 assume il valore $h_4 = 110$. La nuova matrice di trasformazione dello stato corrente dopo l'applicazione della funzione f_2 è:



Al valore corrente della variabile HK viene applicato uno shift verso sinistra di 3 posizioni diventando così pari a 100100000000 e, in seguito, aggiunta la 4^a parte della chiave di Hilbert calcolata (h_4); il valore della variabile HK dopo la quarta ed ultima iterazione è HK = 100100000110_2.

Al termine dell'algoritmo la chiave di Hilbert, generata a partire dai tre attributi di input, è dunque pari a HK = 100100000110_2.

4.2 Z-order curve

La generazione della chiave derivata sulla curva *Z-order* è molto più semplice rispetto a quella sulla curva di Hilbert e si effettua tramite una manipolazione dei bit delle rappresentazioni binarie del valore degli n attributi.

4.2.1 Generazione della chiave derivata sulla curva Z-order

L'algoritmo utilizzato per la generazione della chiave derivata di un punto sulla Z-curve, a partire da n attributi numerici, è ripreso dall'articolo [24]. Lo scopo dell'algoritmo è quello di determinare un'associazione univoca tra il valore di n attributi ed un punto appartenente alla curva *Z-order*, le cui coordinate compongono la chiave derivata o *Z-Curve derived-key*. Si effettua dunque un passaggio da un sistema n-dimensionale formato dagli n attributi ad un sistema lineare formato dalle chiavi derivate sulla *Z-order Curve*.

La chiave derivata sulla *Z-curve* è composta da $n \times k$ bits, dove n rappresenta il numero di dimensioni della curva (pari al numero di attributi) e k il numero di bit utilizzati per la codifica binaria del valore di ogni singolo attributo; per esempio, nel caso in cui, il valore degli attributi sia contenuto nel range [0 - 31], saranno necessari 5 bit per la rappresentazione binaria del valore e dunque k sarà pari a 5. La costruzione della chiave derivata prevede l'interleaving dei bit che compongono la codifica binaria dei valori degli n attributi ed è effettuata concatenando il valore dell'i-esimo bit di ciascun attributo a partire dal bit più significativo, quindi a partire da $i = 0$, fino ad $i = k-1$. Supponiamo di avere n attributi $A^0 ... A^{n-1}$ di valore compreso nell'intervallo $[0\,;2^k - 1]$, la chiave derivata sarà composta da $n \times k$ bit ed assumendo $a_0^i \ldots a_{k-1}^i$ la

rappresentazione binaria del valore dell'attributo i-esimo, la chiave derivata risultante sulla curva Z-order, ZK, sarà così formata:

$$A^0 = a_0^0\, a_1^0 .... a_{k-1}^0$$
$$\vdots$$
$$A^{n-1} = a_0^{n-1}\, a_1^{n-1} .... a_{k-1}^{n-1}$$

$$\text{ZK} = a_0^1 a_0^0 \ldots a_0^{n-1} a_1^1 a_1^0 \ldots a_1^{n-1} \ldots a_{k-1}^1 a_{k-1}^0 \ldots a_{k-1}^{n-1}$$

4.2.2 Esempio di generazione della chiave derivata sulla curva Z-order

Supponiamo di avere due attributi A^1e A^0 di valore rispettivamente 3 e 5 e che l'ordine k della curva sia pari a 4, quindi con attributi che possono assumere valori compresi nell'intervallo $[0\,;15]$. La codifica binaria di A^1 è pari a 0011 (va espressa su k bit), mentre quella di A^0 è 0101. La costruzione della chiave derivata ZK di $n \times k = 2 \times 4 = 8$ bits è data dall'interleaving dei bit delle due chiavi derivate e procede come indicato in figura 4.7.

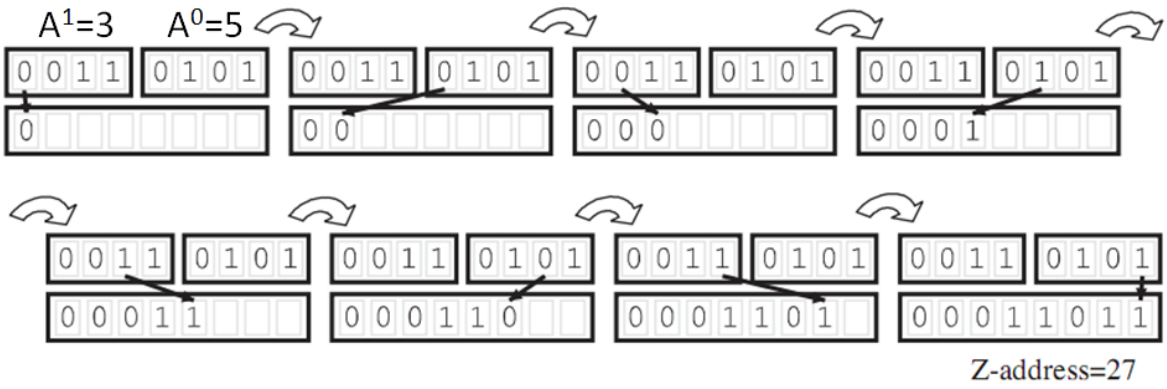


Figura 4.7 – Costruzione della chiave derivata sulla curva Z-order 2D

La chiave derivata risultante ZK avrà una codifica binaria pari a 00011011_2 di $n \times k = 2 \times 4 = 8$ bits ed un valore in rappresentazione decimale pari a 27.

4.3 Complessità degli algoritmi di generazione della chiave derivata

I due algoritmi di generazione della chiave derivata di Hilbert e della curva Z-order sono molto diversi tra loro. Se l'algoritmo per il calcolo della chiave derivata per la curva Z-order risulta semplice ed immediato in quanto richiede solamente una fusione dei bit dei vari attributi, l'algoritmo per la generazione della chiave derivata di Hilbert al contrario è molto più complicato e richiede il calcolo preliminare di una tabella di generazione degli stati ed il calcolo ad ogni iterazione di due funzioni particolari, per determinare la matrice dello stato successivo della curva ed una sezione di n bits della chiave derivata.

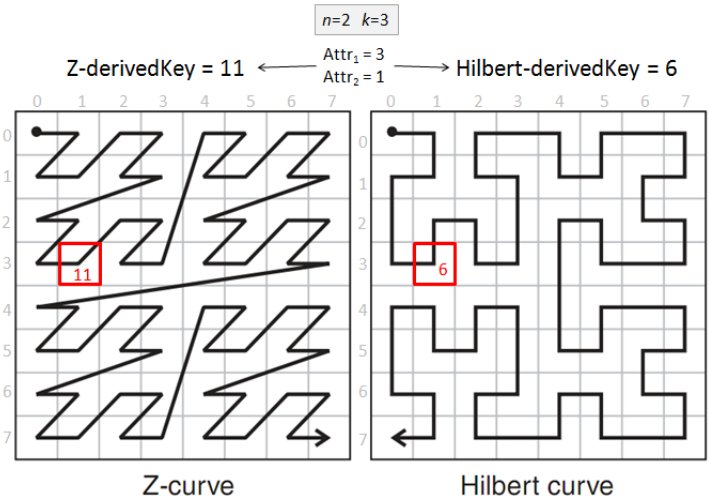


Figura 4.8 – Mapping di una risorsa mediante curve space-filling diverse

L'algoritmo per il calcolo della chiave derivata sulla curva Z-order esegue una scansione lineare degli $n \times k$ bits degli n attributi fondendoli in un'unica chiave di $n \times k$ bits. La complessità della generazione della chiave derivata per la *Z-order curve* è $O(k \cdot n)$. E' possibile definire un'ottimizzazione per l'algoritmo basata sull'interleaving dei bits, ma tale ottimizzazione diminuisce il tempo di esecuzione dell'algoritmo di un *fattore costante* e quindi non ne modifica la *complessità*.

L'algoritmo per il calcolo della chiave derivata sulla curva di Hilbert si compone di due parti: la prima parte riguarda il calcolo della tabella di generazione degli stati o

generator table che può essere eseguito una volta soltanto e utilizzata successivamente per la generazione di tutte le chiavi derivate necessarie. La seconda parte dell'algoritmo riguarda il calcolo vero e proprio della chiave derivata di Hilbert, formata da $n \times k$ bit. Questa parte richiede k iterazioni (con k pari all'ordine di raffinamento della curva) in ognuna delle quali si calcolano le due funzioni f_1 e f_2 (si veda la sezione 4.1.2). La funzione f_1 è composta da due step successivi: il primo prevede il calcolo dell'or esclusivo (XOR) tra il valore corrente della variabile z_i e l'array *signs,* che contiene i segni della matrice di trasformazione dello stato corrente, mentre nel secondo si effettua una permutazione dei bit del valore ottenuto come risultato del primo passo a seconda del valore degli elementi della matrice matr_{Sc} corrente. La funzione f_2, come la funzione f_1 si compone anch'essa di due step successivi: nel primo si effettua il calcolo della matrice di trasformazione per il prossimo stato (matrice A), come permutazione delle righe della matrice di trasformazione per lo stato successivo ricavata dalla *generator table* al passo precedente, in base al valore degli elementi della matrice di trasformazione dello stato corrente attuale. Il secondo step della funzione f_2 modifica i segni degli elementi della matrice A in maniera differente a seconda di alcuni casi particolari dati dalle combinazioni dei segni delle matrici dello stato corrente attuale e di trasformazione per lo stato successivo, ricavata dalla generator table al termine del calcolo della funzione f_1.

Analizziamo quindi la complessità dell'algoritmo di generazione della chiave derivata per la curva space-filling di Hilbert, definito in figura 4.5. L'algoritmo prevede un loop più esterno di k *iterazioni*, tante quanto è l'ordine della curva space-filling. All'interno di questo loop viene eseguito un certo insieme di operazioni di costo $O(n)$. Ad esempio la costruzione della variabile z_i al primo passo richiede la concatenazione dei bit corrispondenti delle rappresentazioni degli n attributi. *La complessità dell'algoritmo è quindi* $O(n \cdot k)$.

La complessità di generazione delle chiavi derivate per i due tipi di curva *è quindi la stessa.* Questa complessità è infatti inerente al problema della generazione della chiave derivata che richiede un numero di iterazioni pari al numero di *suddivisioni dello spazio* e quindi dell'*ordine della curva.* Per ogni iterazione occorre elaborare *1 bit per ognuna delle coordinate.* Si noti però che, a differenza della Z-order curve, la generazione della chiave derivata per la curva di Hilbert richiede l'esecuzione di operazioni di complessità *costante ad ogni iterazione.* La differenza tra gli algoritmi risiede quindi nella complessità della loro implementazione e nell'*effetto dei fattori*

costanti sul tempo di esecuzione. Si noti inoltre che il tempo di esecuzione di entrambi gli algoritmi è proporzionale al numero di bit richiesti per la rappresentazione della chiave derivata. C'è inoltre da considerare che l'algoritmo per la generazione della chiave di Hilbert richiede la generazione della *generator table* e che questa può risultare di *grande dimensione per valori di n e k elevati*. Il numero di righe di questa matrice risulta pari al numero di chiavi derivate su una curva del primo ordine e quindi 2^n. Ogni riga contiene nelle colonne Y, X_1 ed X_2 valori di n bits ed una matrice $n \times n$. Si ottiene quindi una dimensione della matrice di $O(2^n \cdot 2^n)$ bits. Per valori elevati di n la dimensione della tabella può assumere quindi dimensioni elevate. Inoltre, come vedremo nel capitolo 6, gli algoritmi per la risoluzione di query sulla curva Z-order sono estremamente più semplici. Queste limitazioni hanno portato alla scelta della *Z-order curve per l'implementazione delle range query multiattributo in HASP*.

Capitolo 5

HASP: l'architettura

In questo capitolo presentiamo l'architettura generale di *HASP*. *HASP* definisce un albero di agggregazione. La struttura dell'albero e le operazioni su di esso sono riprese da *XCone*. *HASP* estende *XCone* per il supporto al caso multidimensionale. Questo implica la necessità di inserire la generazione della chiave derivata e cambiare le strutture di aggregazione ed il metodo di risoluzione delle range query. Nelle prossime sezioni verrà prima descritto l'albero *XCone* e successivamente mostrate le strategie di aggregazione definite per *HASP.*

5.1 XCone

XCone [44] è una rete P2P che nasce come estensione del sistema Cone descritto in precedenza il quale supporta solamente query del tipo *Trova k risorse maggiori di S.* In XCone diversamente, grazie ad una struttura ad albero distribuita, costruita al di sopra di una qualunque DHT, è possibile definire una strategia di aggregazione delle risorse tale per cui risulti efficiente risolvere query del tipo *Trova k risorse per cui il valore X sia compreso tra* $V \leqslant X \leqslant S$.

5.1.1 L'albero di mapping

In XCone l'aggregazione delle risorse è garantita dall'utilizzo di un albero binario, che offre la possibilità di sfruttare diverse strategie a seconda della funzione di mapping adottata, basato sullo spazio degli indentificatori della DHT.

Sono definiti *nodi logici* quei nodi dell'albero XCone che risultano essere mantenuti in modo distribuito tra i *nodi fisici* o *peer* della rete. I nodi logici foglia sono associati in modo univoco ai nodi fisici attraverso un processo di integrazione con la DHT che sarà illustrato in seguito, mentre, per quanto riguarda i nodi logici non foglia, l'associazione con i nodi fisici è effettuata per mezzo di una funzione di mapping. Il numero di nodi logici non foglia da associare a ciascun peer è determinato per mezzo di tale funzione di mapping e, nel caso pessimo, il numero di nodi logici assegnati ad un peer è inferiore ad un valore h pari al cammino dalla foglia alla radice dell'albero. La funzione di mapping ha un ruolo fondamentale nella costruzione dell'albero di XCone, in quanto a partire da due nodi logici a livello $l-1$ gestiti da due peer distinti, decide a quale dei due nodi assegnare la gestione del nodo logico di livello l.

In linea teorica, ad un qualsiasi peer può essere associato un qualsiasi nodo logico, ma per ottenere buone prestazioni in fase di entrata di un nodo nella rete (*join*) e di risoluzione delle query è necessario imporre alcuni vincoli. Ciascun peer è identificato da un ID univoco in formato binario e la sua posizione all'interno dell'albero è determinata in base al valore di tale identificatore. Ogni peer può gestire solo nodi logici che abbiano un ID tale da avere un prefisso comune ad esso. Questa regola di assegnazione dei nodi logici ai peer permette di implementare in modo efficiente la fase di *join,* in quanto il peer, che si unisce nell'albero XCone per la ricerca dei nodi logici da gestire, deve risalire l'albero a partire dalla foglia assegnata, evitando di visitare diversi sotto-alberi. Da ciò si deduce come l'assegnamento dei nodi foglia ai peer sia univoco, mentre, per quanto riguarda i nodi logici interni, i quali condividono il proprio prefisso con più nodi foglia, essi possono essere assegnati ad uno qualunque dei peer che corrispondono a tali foglie. Un ulteriore vincolo è dato dal fatto che il generico nodo P gestisca un insieme di nodi logici, che corrispondono ad un suffisso del proprio identificatore (ID). Il nodo P gestisce dunque una sequenza continua di nodi logici a partire dalla foglia da lui gestita, fino ad un certo livello l dell'albero in modo da ridurre al minimo il numero di *hops* tra i diversi peer in fase di ricerca durante la risalita dell'albero XCone.

In XCone è definita un'altra funzione: *la funzione di digest.* Tale funzione associa a ciascun nodo una struttura in grado di sintetizzare le chiavi contenute nel sotto-albero radicato in esso. Esistono diversi tipi di funzioni di mapping in grado ciascuna di valutare aspetti diversi, quali il carico dei nodi logici gestiti da ogni nodo o la

tipologia di banda. Ad esempio è possibile adottare una tecnica per migliorare il bilanciamento del carico in modo da regolare il numero di entrate significative presenti nella *routing table* di ciascun nodo. Nel momento in cui un nuovo nodo P entra in XCone, controlla se nel cammino che lega la foglia ad esso associata e la radice esistono nodi logici mappati a peer carichi. In tal caso il nodo P prende in gestione alcune entrate della tabella di routing di tali nodi. Questa operazione è naturalmente seguita da un aggiornamento del mapping di alcuni logici ai peer presenti sul cammino di P dalla foglia alla radice.

Nel caso di XCone la funzione di mapping adottata coincide con quella di Cone basata sulla scelta del peer avente in gestione la chiave di valore massimo ma, come vedremo in seguito, XCone definisce funzioni di digest diverse rispetto a Cone per supportare più efficacemente le range query.

L'assegnamento tra i peer della rete e i nodi foglia XCone viene realizzato a seguito dell'integrazione con la rete DHT sottostante. Se consideriamo una rete DHT con spazio degli identificatori (ID) di m bit, i nodi logici foglia sono assegnati ai peer attraverso un *trie*.

A ciascun peer in fase di *join* viene associato un identificatore di m bit in maniera casuale e ad ogni peer può essere associato un unico nodo foglia all'interno del *trie* in quanto l'albero XCone definisce un *trie* basato sui prefissi degli identificatori.

La figura 5.1 illustra, attraverso un esempio, il mapping tra nodi logici foglia dell'albero XCone ed i peer di una rete DHT Chord con identificativi di 4 bits.

L'associazione nodo/foglia rimane fisssata per tutta la permanenza del peer in rete. L'identificatore di ogni nodo è generato tramite una funzione hash impiegata dalla DHT per assegnare i peer alla rete e ciascun nodo gestisce in maniera indipendente la propria chiave, la quale può variare e non comporta uno spostamento del nodo all'interno del'albero. Il dominio delle chiavi e quello degli identificatori risulta quindi essere distinto.

5.1.2 La tabella di routing

La tabella di routing è la struttura che mantiene l'albero di mapping distribuito tra i peer della rete. In tale struttura vengono memorizzati, per ciascun nodo, una lista di nodi da esso gestita. La routing table è composta al massimo da m entrate, una per

ogni livello dell'albero, a partire dal livello 0.

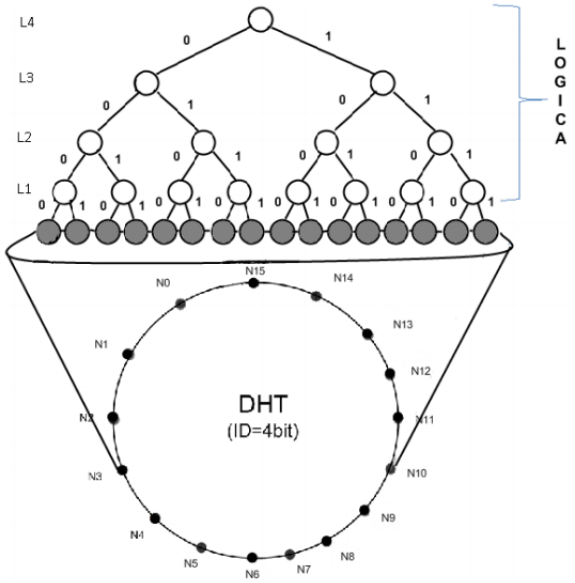


Figura 5.1 – Overlay XCone-DHT

La generica entrata E_l memorizza il risultato dell'applicazione della funzione di mapping f tra due nodi logici adiacenti di livello $l-1$. A tale posizione corrisponde una coppia di valori (IP_1, IP_2), avente il seguente significato:

- IP_1: indirizzo del peer che gestisce il nodo logico di livello l.
- IP_2: indirizzo del peer che gestisce il nodo logico assegnato come figlio di livello l-1.

La mancanza di un limite inferiore alle chiavi contenute nel sotto-albero o più in generale la mancanza della conoscenza della distribuzione delle chiavi, potrebbe portare ad una visita infruttuosa o ad una perdita di nodi target nel caso di esplorazione o meno del sotto-albero. Per questo è stato necessario ampliare le informazioni rispetto a Cone, aggregando nel migliore dei modi le informazioni sulle chiavi presenti nei sottoalberi. La generica entrata della Routing Table E_l viene estesa tramite l'aggiunta del seguente campo:

- ST, contiene l'informazione di digest ovvero i valori delle chiavi presenti nei peer associati al sotto-albero radicato nel nodo figlio.

Da notare come le informazioni memorizzate siano dei riferimenti diretti ad indirizzi di rete dei peer. Questa è una scelta progettuale che consente di evitare un ulteriore livello di indirezione rappresentato dal passaggio per la rete DHT. In altri termini vengono velocizzate le operazioni di comunicazioni evitando inutili ricerche nella rete DHT.

5.1.3 Le operazioni

Nei due paragrafi seguenti saranno esaminate le operazioni di join e di ricerca effettuate da un nodo fisico in XCone.

Operazione di join

Per descrivere l'operazione di *join* di un nodo sulla rete XCone consideriamo l'esempio in figura 5.2. Il peer P_0 esegue l'operazione di join nella rete DHT e in questa fase ottiene l'assegnamento dell'ID e del relativo nodo logico foglia di XCone. Contestualmente all'ingresso in rete, la DHT invia al peer P_0 un riferimento ad un nodo già presente in rete, nell'esempio al nodo P_1, in modo tale da condividere con P_0 il prefisso più lungo dell'ID. Formalmente P_1 viene definito il *Least Common Ancestor (LCA)* di P_0.

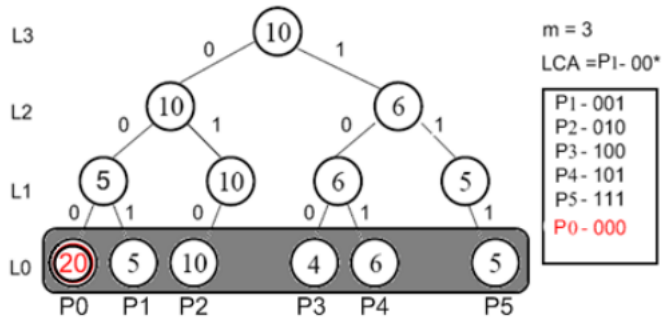


Figura 5.2 – Esempio di *join* di un peer nella rete XCone

Prima di continuare con l'esempio descriviamo più in dettaglio il concetto di *LCA* e la sua localizzazione. In generale quanto un peer P si unisce alla rete XCone ottiene mediante la DHT un identificatore ID. Supponiamo che al momento dell'entrata di P nel sistema, sulla DHT e quindi in XCone, siano già presenti N nodi con identificatori $ID_1, ..., ID_i, ..., ID_N$. E' necessario che P individui tra questi nodi un nodo M il cui

identificatore ID_m possieda la più lunga porzione di prefisso in comune con ID. E' possibile che esistano diversi nodi con questa caratteristica. Il nodo *LCA* può essere individuato mediante la DHT, infatti è possibile dimostrare che il predecessore o il successore di *P* sulla DHT è uno dei nodi il cui identificatore condivide con ID il più lungo prefisso. L'operazione di *lookup*, implementata in ognuna delle DHT attualmente esistenti, consente di reperire il successore ed il predecessore di un nodo e può essere utilizzata per supportare la ricerca del *LCA* in XCone. Il peer *P* ottiene dunque tramite la DHT il riferimento al suo predecessore ed al suo successore e sceglie tra questi quello con cui condivide il più lungo prefisso dell'identificativo.

Si consideri, ad esempio, la figura 5.3. Si può notare come il nodo *N1* sia quello che condivide il più lungo prefisso con *N2*, mentre il nodo *N3*, pur essendo adiacente ad *N2* risiede dalla parte opposta dell'albero XCone e pertanto non condivide alcun prefisso con *N1*.

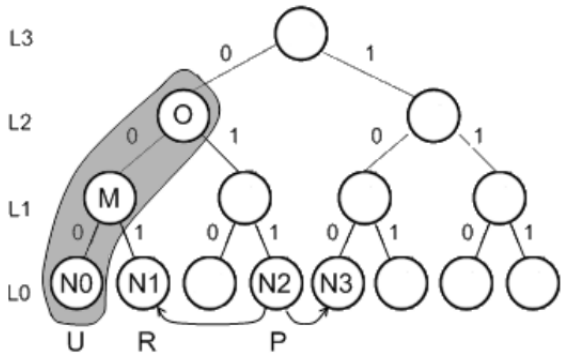


Figura 5.3 – Localizzazione del *LCA* e del nodo logico di intersezione in XCone

Una volta individuato il peer *LCA,* viene calcolato il nodo logico di intersezione tra i due cammini, nell'esempio rappresentato dal nodo *O*. Il nodo *P* invia un messaggio di join ad *N1* specificando anche il livello L3 del nodo di intersezione *O*. Questo ulteriore parametro viene inserito in quanto, come nell'esempio riportato in figura 5.3, il peer *R* a cui è associato il nodo logico *N1*, a seguito di un nuovo mapping dei nodi, potrebbe non avere più in gestione il nodo logico *O*.

La fase di *join* individua, attraverso la DHT, il nodo *LCA* con il procedimento descritto precedentemente. In seguito, attraverso il calcolo del nodo logico di intersezione, il nodo fisico *LCA* provvede a verificare se risulta ancora essere il gestore del nodo logico di intersezione e, nel caso in cui non lo fosse, provvede ad inoltrare

la richiesta di join al proprio nodo padre. Questo procedimento esegue una risalita dell'albero lungo il cammino che porta al peer avente in gestione il nodo logico di intersezione. A questo punto l'operazione di *join* può continuare regolarmente con le ulteriori operazioni necessarie.

Riprendendo l'esempio iniziale riportato in figura 5.2, sono stati indicati in basso i peer assegnati ai rispettivi nodi logici foglia, mentre all'interno di ogni nodo logico è riportata la chiave gestita dal peer a cui il nodo logico è stato associato. Una volta assegnato il nuovo ID ed identificato il peer LCA, ha inizio la fase di risalita bit a bit o di *trickling*.

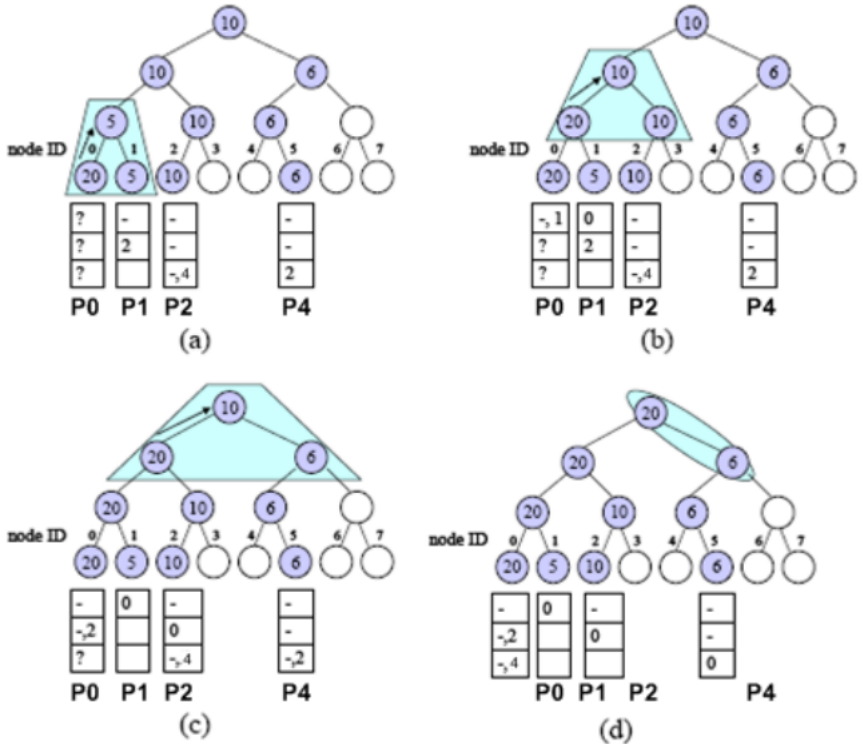


Figura 5.4 – Esempio fase di *trickling* in XCone

P_0 confronta il proprio ID con quello di P_1 determinando in questo modo il livello di partenza, $l = 1$ (fig. 5.4(a)). P_0 possiede un valore maggiore rispetto a quello gestito da P_1 e pertanto procede con l'inserire nella propria tabella di routing l'entrata $(-, \mathrm{IP}_{P1}, d(P_1))$ (fig. 5.4(b)). Contestualmente viene informato anche il nodo P_1 sul nuovo padre. P_1 aggiornerà di conseguenza la sua routing table, inserendo al livello 1 l'entrata (IP_{P0}) e rimuovendo le informazioni ai livelli successivi (fig. 5.4(b) e (c)).

Continuando la risalita al livello 2, P_0 determina il nodo con cui confrontarsi grazie alle informazioni recuperate dalla tabella di routing del nodo P_1. Si effettua un confronto con P_2 e anche in questo caso il nodo P_0 risulta essere il padre del livello 2. P_0 procede esattamente nello stesso modo aggiornando la tabella di routing al livello 2 con il valore $(-, \mathrm{IP}_{P2}, d(P_2))$ ed informando P_2 sul cambio di padre (fig. 5.4(c)). La fase di *trickling* termina al terzo livello rilevando nuovamente un aggiornamento dell'albero e quindi una variazione nelle routing tables (fig. 5.4(d)).

Analizziamo adesso la complessità dell'operazione di join di un nodo nella rete XCone. La fase di inserimento nella DHT e di ricerca del nodo *LCA* viene effettuata dal livello sottostante con costo al caso pessimo di *O(log T)*, con T numero massimo di peer. La fase di *trickling*, come visto nell'esempio, può impiegare al caso pessimo *O (log T)* operazioni pari al numero di peer presenti nel percorso dalla foglia alla radice dell'albero. L'operazione di *join* risulta dunque avere un costo complessivo logaritmico nel numero di peer.

Operazione di ricerca (Find)

L'operazione di ricerca può essere eseguita a partire da un qualsiasi peer P della rete. Consideriamo il caso in cui un peer P voglia effettuare una query Q, in cui richiede K nodi con valore X compreso nell'intervallo $V \leqslant X \leqslant S$. La ricerca consiste in un'esplorazione dell'albero XCone mediante le informazioni contenute nella tabella di routing dei vari peer. La procedura di ricerca può essere suddivisa in due fasi. La prima fase consiste nella risalita dell'albero XCone denominata anche operazione di risalita o di *trickling*, mentre la seconda fase nell'esplorazione dei sotto-alberi. In XCone la prima operazione viene effettuata in maniera sequenziale, in particolare l'esplorazione di ogni sotto-albero, rappresentato da un'entrata della tabella di routing, viene eseguita controllando mano a mano che il numero di risorse localizzate sia sufficiente e in tal caso il procedimento di risalita viene interrotto. Nel processo di esplorazione dei sotto-alberi la ricerca avviene in parallelo guidata dalle informazioni di digest: dopo aver verificato tramite l'informazione di digest la presenza di valori in grado soddisfare i vincoli imposti dalla query, un nodo propaga immediatamente la richiesta di esplorazione in basso verso i nodi foglia.

Supponiamo che il generico peer P riceva una query Q, in cui si richiedono K risorse e che la query sia stata originata dal peer denominato *QueryNode*. Il generico peer P utilizzando una funzione *matchValue* controlla se la propria chiave soddisfa

la query Q. In caso affermativo il nodo P invia il proprio indirizzo come risultato al *QueryNode*. Il *QueryNode* tiene traccia dei risultati raccolti e provvede a trasmettere al peer P il numero di peer che attualmente soddisfano i vincoli imposti dalla query Q. Il peer P pertanto analizza il valore ricevuto dal QueryNode, *collectedMatches,* e, se risulta già raggiunto il numero di risorse K richieste, termina l'operazione attraverso un'uscita forzata. Supponendo che le risorse localizzate non siano sufficienti, il peer P procede a collezionare un insieme S di propri sotto-alberi attraverso le informazioni possedute nella propria routing table. La funzione di digest consente di selezionare esclusivamente i sotto-alberi in cui è stata rilevata la presenza di risorse compatibili con la query Q. Per tali sotto-alberi, in maniera sequenziale, il nodo P invia una richiesta di esplorazione al peer che gestisce la radice del sotto-albero. Una volta inviato il messaggio, prima di passare alla successiva esplorazione, il peer P attende di ricevere una risposta dal *QueryNode* sul numero di risorse localizzate nella precedente richiesta. In questo modo, se il numero di risorse dovesse risultare sufficiente, viene interrotta la fase di esplorazione dei successivi sotto-alberi e di conseguenza terminata la fase di *trickling* attraverso un'uscita forzata. Questa operazione di sospensione dell'eplorazione ha come fine quello di ridurre al numero strettamente necessario il numero di nodi esplorati nell'albero XCone. Terminata la fase di esplorazione il peer P controlla se non sono stati esplorati sotto-alberi, in questo caso effettua direttamente l'operazione di trickling inviando un messaggio di richiesta al proprio nodo padre. Nel caso in cui siano stati esplorati i sotto-alberi, il nodo P sospende la ricerca inviando un messaggio di trickling al *QueryNode* ed attende nuovamente come risposta il numero di risorse localizzate. Anche in questo caso la sospensione è effettuata al fine di controllare la propagazione verso il nodo radice delle operazioni di ricerca. Per quanto riguarda le operazioni di esplorazione dei sotto-alberi, il nodo radice R controlla preliminarmente se la propria chiave soddisfa la query Q. In tal caso invia il proprio indirizzo al *QueryNode* e successivamente riceve da quest'ultimo il numero di risorse localizzate. Nel caso in cui le risorse non siano sufficienti, R inoltra in parallelo le richieste di esplorazione dei propri sotto-alberi. La procedura di esplorazione procede verso i nodi logici foglia dell'albero XCone e viene diretta ai soli nodi foglia che, attraverso le informazioni di digest, risultano compatibili con la query Q. L'algoritmo procede nella maniera più veloce a recuperare le relative risorse.

Il costo di una operazione di ricerca in XCone è determinata dal costo della fase

di *trickling*. Tale fase, come osservato, può coinvolgere nel caso pessimo un intero cammino dalla foglia al nodo radice dell'albero, pertanto al massimo *O (log T)* nodi fisici, con T numero di peer nella rete. Il numero totale di messaggi ricevuti dal *QueryNode*, avendo una esplorazione sequenziale dei sotto-alberi, è dato da al più K messaggi, corrispondenti ai risultati ricevuti, a cui vanno aggiunti *log T* messaggi di richieste di *trickling* per un totale di *O (log T + K)* messaggi ricevuti.

5.2 Aggregazione di chiavi derivate: strutture di digest

Le funzioni di digest permettono di aggregare un insieme di valori in un'unica informazione che richiede spazio in memoria limitato e che cattura nel migliore dei modi l'informazione sulla distribuzione delle chiavi. In questa sezione verrano esaminate le tecniche di digest *BitVector* e *q-digest*, come sono utilizzate in *XCone* [44] e come sono state modificate per essere impiegate in *HASP* (*Hierarchical Aggregation over Space-filling*).

5.2.1 BitVector Indexes

La prima funzione di digest definita in *HASP* è il *BitVector Index*.

5.2.1.1 BitVector Indexes con intervalli equidistanti (eBV)

Al fine di adattare la tecnica di digest *BitVector Indexes* al nuovo range di valori dello spazio delle chiavi derivate ($\left[0; 2^{n \times k} - 1\right]$ con n uguale al numero di attributi e k pari al numero di bit da utilizzare per la codifica binaria del valore di ciascun attributo), una soluzione possibile consiste nel dividere l'intervallo dei possibili valori delle chiavi derivate in un numero h di intervalli tutti della stessa ampiezza. Il numero h degli intervalli rappresenta il numero di elementi del vettore *BitVector* in cui ogni elemento indica se almeno una chiave derivata cade in tale intervallo o meno. Il numero di intervalli con cui dividere lo spazio totale delle chiavi derivate, come detto, influenza il numero di elementi del vettore *BitVector*, e risulta pertanto limitato.

Definiamo un vettore binario di k bits *BitIndex* in maniera tale da catturare la distribuzione delle chiavi derivate A, definito in un intervallo [a, b], nel seguente modo.

Partizioniamo il dominio delle possibili chiavi derivate, A, in modo da ottenere h-1 intervalli di separazione $a = a_0 < a_1 < .... < a_h = b$. Per ogni possibile istanza di $\{A = v\}$, il vettore binario *BitIdx* di h bits è definito nel seguente modo:

$$\text{BitIdx}(A) = \begin{cases} 1, & \text{se } v \in [a_j, a_{j+1}] \quad \text{con } 0 \leqslant j \leqslant h-1 \\ 0, & \text{altrimenti} \end{cases}$$

Il *BitVector Index* è calcolato come l'unione (OR) di tutti i rispettivi *BitIdx* in modo da ottenere un unico vettore di h posizioni che esprime la presenza o meno di una o più chiavi derivate all'interno dell'i-esimo *bucket*. Se dal punto di vista teorico tale soluzione risulta utilizzabile, nella pratica, la divisione in un numero limitato di intervalli di uno spazio delle chiavi di ampiezza molto elevata, ad esempio dell'ordine di 2^{60} elementi, conduce ad un'approssimazione troppo grande, tale da rendere la tecnica di digest inefficace. Infatti le chiavi derivate potrebbero essere raggruppate in cluster contenenti valori vicini tra loro, con differenze sensibilmente minori rispetto all'ampiezza del singolo intervallo. D'altra parte la definizione di un numero di intervalli troppo grande, dello stesso ordine del valore delle chiavi, può provocare problemi di allocazione in memoria del vettore *BitVector*. L'utilizzo di tale strategia di digest risulta tanto più inefficace, quanto minore è il numero di intervalli in cui è diviso lo spazio delle chiavi originario. Per esemplificare il funzionamento di tale tecnica di digest, in figura 5.5 è riportato un esempio della divisione in 4 intervalli di uno spazio delle chiavi di dimensione 16.

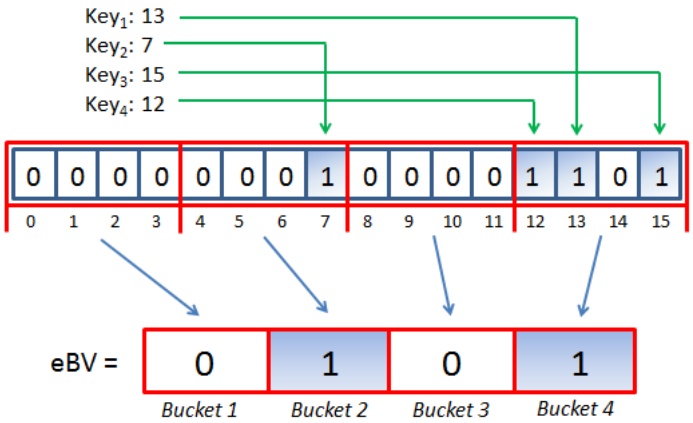


Figura 5.5 – Esempio di costruzione di *BitVector* con intervalli di uguale dimensione

E' possibile notare come, per le chiavi specificate, il *BitVector* risultante fornisca già un'approssimazione decisamente grande nonostante lo spazio limitato delle chiavi. Sia per il *bucket* numero 2 che per il numero 4 viene espressa, dal *BitVector*, la presenza di almeno una chiave derivata che cade in tali intervalli, ma non vi è alcuna indicazione sul fatto che nell'intervallo numero 4 sia presente un numero di risorse tre volte superiore rispetto a quelle contenute nell'intervallo numero 2. L'applicazione della tecnica *eBV*, all'interno di *HASP*, introduce una grande approssimazione sulla distribuzione delle chiavi su un numero limitato k di intervalli specificato dall'utente. Tale approssimazione influisce a sua volta sul numero di nodi visitati dell'albero *HASP* per la risoluzione di range query multidimensionali. La tecnica dell'*eBV* è stata comunque testata in *HASP*, in modo da fornire un termine di paragone con le tecniche di digest che verranno presentate in seguito.

5.2.1.2 Data driven BitVector (dBV)

In questa sezione verà presentata una variante della tecnica di digest *BitVector Indexes, applicabile solamente nel caso in cui si conosca a priori la distribuzione delle chiavi derivate all'interno dello spazio delle risorse.* In tal caso, è possibile definire gli intervalli di divisione dello spazio delle chiavi derivate, nel *BitVector*, direttamente a partire dal valore di esse. Un esempio del *BitVector* risultante dall'applicazione di questo procedimento di divisione a priori dello spazio delle chiavi è mostrato in figura 5.6.

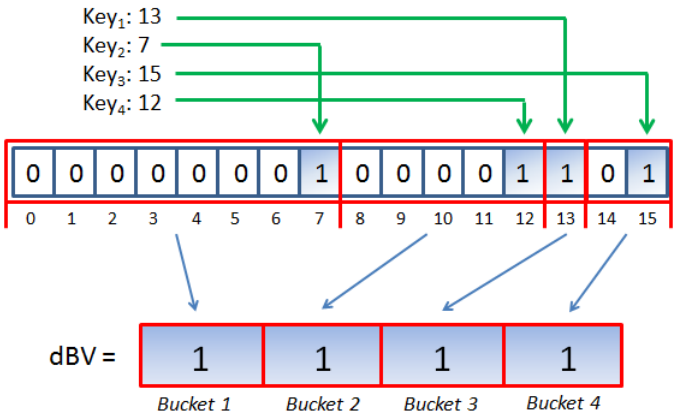


Figura 5.6 – Esempio di costruzione del BitVector con intervalli guidati dalla conoscenza a priori del valore delle chiavi

E' possibile notare come, per le risorse specificate, il *data-driven BitVector* (dBV) risultante fornisca un'approssimazione minore della distribuzione delle chiavi derivate rispetto alla variante con intervalli di ampiezza costante, eBV. Nel vettore costruito, il numero di elementi è rimasto uguale rispetto alla versione eBV, ma è cambiato il mapping tra gli elementi e gli intervalli a cui sono riferiti. In questa variante dBV, gli intervalli non possiedono tutti la stessa ampiezza, ma sono definiti in maniera tale da determinare una distribuzione quanto più uniforme possibile delle chiavi derivate. Rispetto alla versione eBV, in questa variante in cui gli intervalli sono guidati dal valore delle chiavi, si ha un minor sbilanciamento sul numero di chiavi derivate che cadono nei vari bucket. Il problema del numero di intervalli limitato presente nella versione eBV rimane tuttavia presente anche in questa variante seppur in maniera meno importante, in quanto affinchè si presenti è necessario che il numero di chiavi derivate del sistema sia estremamente grande: il vettore dBV possiede tanti elementi quante sono il numero di chiavi. Ad ogni modo, se si vuole utilizzare un numero di intervalli k, minore del numero delle risorse inserite, è possibile, per prima cosa definire tutti gli intervalli del *BitVector* senza tener conto del valore di k e successivamente fondere alcuni intervalli scelti in maniera *round-robin* fino a raggiungere il numero di intervalli k richiesto. In figura 5.7 è mostrata la compressione su 2 intervalli del vettore dBV dell'esempio di figura 5.6: l'intervallo numero 1 viene unito all'intervallo numero 2 per creare un unico bucket e la stessa cosa avviene per i bucket 3 e 4. Ovviamente un minor numero di intervalli aumenta l'approssimazione sulla distribuzione delle chiavi derivate, rendendo meno uniforme il numero di risorse che cadono in ciascun bucket.

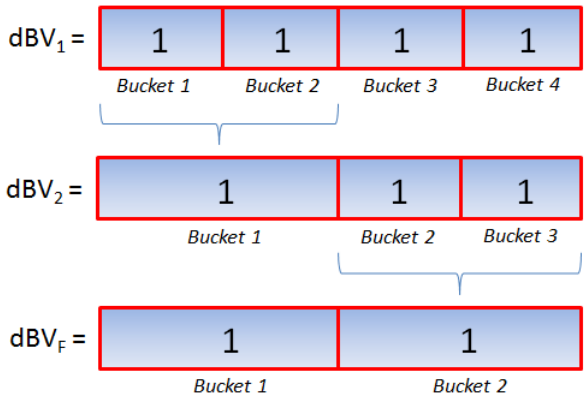


Figura 5.7 – Esempio di compressione del BitVector aBV di figura 5.6 da 4 a 2 elementi

L'applicazione di tale variante *dBV* all'interno di *HASP* permette l'utilizzo di chiavi derivate di grandi dimensioni e non introduce un'approssimazione troppo grande sulla distribuzione delle chiavi su un numero di intervalli k. Tale soluzione è stata testata in *HASP* ed è possibile trovare un confronto tra i risultati ottenuti tra le due varianti *eBV* ed *dBV* nel capitolo 8.

5.2.2 Q-Digest

La seconda funzione di digest utilizzata in *HASP* trae fondamento dal campo delle reti di sensori. Presentata da [40], tale funzione di digest prende il nome di *q-digest*. Nelle reti di sensori si hanno dei vincoli strutturali quali la durata energetica dei sensori, la limitata quantità di memoria o la connessione di rete non sempre affidabile e, per rispondere a queste problematiche, la ricerca si è spinta verso la definizione di soluzioni in grado di sintetizzare nel migliore dei modi i rilevamenti effettuati e di conseguenza ridurre i tempi di trasmissione delle informazioni. La funzione *q-digest* sfrutta le proprietà matematiche dei *quantili*, da cui deriva il nome, per sintetizzare una distribuzione di valori entro un margine di errore fissato. Definiamo un insieme di valori n con dominio definito nel range $[1; \sigma]$. A tale dominio possiamo associare un albero binario T di altezza $\log \sigma$. L'albero T partiziona per ogni livello l'intero dominio attraverso dei range $[min,\ max]$ assegnati ad ogni nodo. In particolare tutti i nodi foglia possiedono un'ampiezza del range pari ad uno, mentre i nodi interni corrispondono all'unione dei range presenti ai nodi figlio di livello inferiore. Ad ogni nodo è associato un contatore di frequenze *count,* che indica il numero di valori presenti nel range. La figura 5.8 mostra un esempio in cui il dominio ha ampiezza $\sigma = 8$ ed il numero totale di valori da sintetizzare è pari a $n = 15$. Il valore n è definito come $n = \sum \text{count}(v), v \in \textit{Leaf Set}$. Una volta definito l'albero *T,* la funzione *q-digest* consiste nel selezionare un insieme di nodi dell'albero, in grado di catturare al meglio la distribuzione dei valori.

Come è facile osservare, la situazione ideale è rappresentata dalla memorizzazione dei nodi foglia, ma, avendo un vincolo di memoria da soddisfare e ipotizzando che l'insieme delle chiavi risulti numeroso, è necessario effettuare una operazione di compressione dell'albero. Fissato un parametro di compressione k, l'operazione di compressione consiste nel raggruppare le chiavi verso i nodi di livello maggiore.

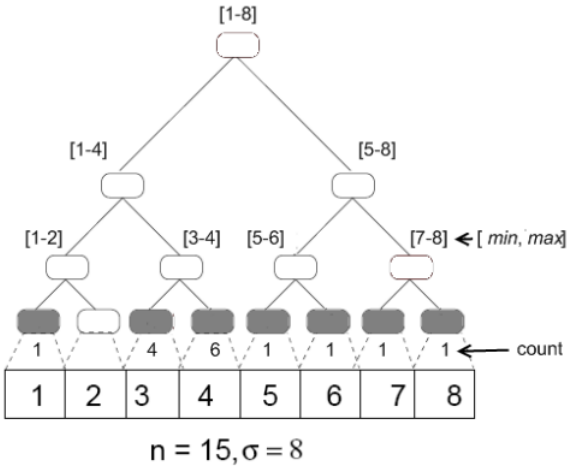


Figura 5.8 – Esempio di albero di *q-digest*

In particolare devono essere soddisfatte le seguenti proprietà (*q-digest properties*):

1. $\text{count}(v) \leqslant \left\lfloor \frac{n}{k} \right\rfloor$
2. $\text{count}(v) + \text{count}(v_p) + \text{count}(v_s) \; > \left\lfloor \frac{n}{k} \right\rfloor$

Il parametro v_p rappresenta il nodo padre ed il parametro v_s il nodo fratello del nodo v. Se il numero di nodi che possiedono chiavi, risulta inferiore o uguale al parametro di compressione k allora l'albero T verrà lasciato intatto e nessuna operazione di compressione verrà applicata, altrimenti sarà necessario applicare tale operazione di compressione dell'albero.

L'algoritmo di compressione verifica che le *q-digest properties* siano rispettate. Nel caso in cui una proprietà risulti violata, l'operazione di compressione procede con l'azzeramento dei contatori del nodo figlio e l'assegnazione al nodo padre del valore dato dalla somma precedente di questi. L'esempio in figura 5.9 mostra tale operazione.

Consideriamo un insieme di valori $n = 15$ appartenenti al range [0,15] e i loro rispettivi contatori di occorrenze (fig. 5.9(a)). Fissando il parametro di compressione al valore $k = 5$ si ottiene un valore soglia di $\left\lfloor \frac{15}{3} \right\rfloor = 3$ utilizzato dalle *q-digest properties*. La figura 5.9(a) mostra come i nodi a, c e d non soddisfino la seconda proprietà di *q-digest*. La compressione procede raggruppando le chiavi nei rispettivi nodi padre, portando alla situazione illustrata in figura 5.9(b). A questo punto, analizzando

l'albero, risulta che il nodo e viola nuovamente la proprietà di digest. Applicando l'operazione di compressione si ha la situazione visibile in figura 5.9(c).

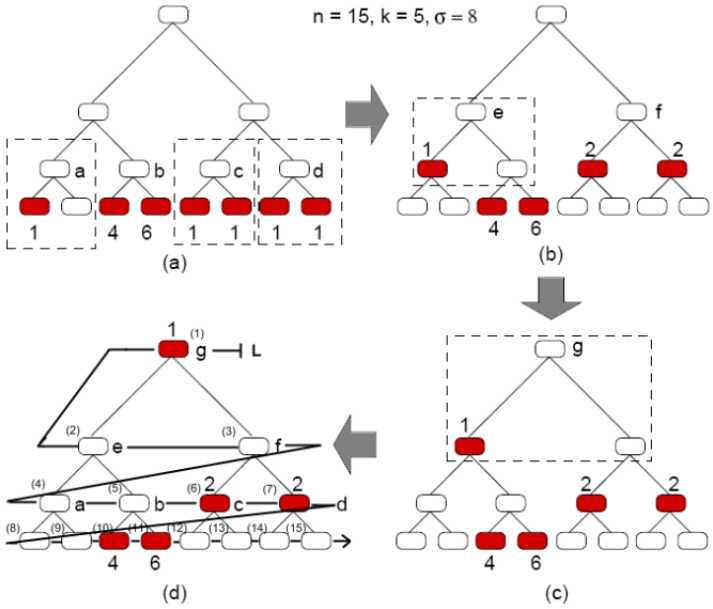


Figura 5.9 – Esempio di compressione dell'albero di *Q-Digest*

Anche in questo caso occorre nuovamente applicare la compressione portando l'albero nello stato finale visibile in figura 5.9(d). Una volta terminata la compressione dei nodi dell'albero T viene definita l'informazione di digest *q-digest* attraverso la rappresentazione compatta dell'albero. Consideriamo i nodi dell'albero T come in una lista L (fig. 5.9(d)) concatenata e numerata da destra a sinistra, dall'alto in basso. Il *q-digest* è quindi composto da un insieme di tuple della forma $(\text{node}_{\text{id}}(v), \text{count}(v))$, tale per cui il contatore di occorrenze di ogni nodo esaminato abbia valore maggiore di zero; nell'esempio in figura 5.9(d) il *q-digest* è quindi rappresentato nel seguente modo:

$$q - \text{digest}(L) = \{(1,1),(6,2),(7,2),(10,4),(11,6)\}$$

Si può vedere come il parametro di compressione k influenzi l'aggregazione dei nodi, in particolare attraverso le proprietà di *q-digest* si mantiene l'operazione di compressione adattiva. In altri termini si tende a mantenere i nodi con range di ampiezza minore, quindi più accurati, nelle zone dell'albero in cui si ha maggiore concentrazione dei dati e ad accorpare nodi con range più ampi i nodi che posseggono dati più sparsi.

Inoltre incrementando i valori da sintetizzare risulta inevitabile un sempre maggiore accorpamento dei nodi in range di ampiezza maggiore.

L'operazione di unione tra più *q-digest* consiste nell'ottenere un unico albero con i contatori di occorrenze di ogni nodo definiti come la somma dei relativi contatori dei singoli alberi *q-digest* e, se necessario, applicare l'operazione di compressione al nuovo albero di *q-digest*.

5.2.3 Strutture di digest: implementazione di BitVector in HASP

Le funzioni di digest utilizzate in *HASP* prendono spunto da quelle utilizzate in XCone, ma entrambe le tecniche di digest, *BitVector Index* e *Q-Digest*, devono essere modificate a causa dell'elevata ampiezza di valori che possono essere assunti dalle chiavi derivate. L'implementazione di *XCone*, realizzata per fornire supporto per attributi unidimensionali, definisce le tecniche di digest *BitVector Index* e *Q-Digest* basandosi sull'assunzione che l'intervallo delle chiavi sia dell'ordine di massimo 10^2 chiavi.

L'utilizzo della tecnica di digest *BitVector* come implementata in XCone non è pensabile per uno spazio delle chiavi derivate talmente ampio come quello necessario nel contesto multidimensionale di *HASP*. Ad esempio, supponiamo di avere uno spazio delle chiavi di dimensione 2^{60}, tipico nel caso multidimensionale dato da n=6 e k=10. Se decidiamo di dividere tale spazio tramite 2^{30} intervalli, è necessario mantenere in memoria un vettore di 2^{30} elementi (1GB elementi). Oltre al problema della grande dimensione del vettore da allocare in memoria, si ha una notevole approssimazione sulla distribuzione delle chiavi derivate sui vari intervalli in quanto ciascun intervallo avrà un'ampiezza decisamente grande e pari in questo caso a 2^{30}. Con intervalli di ampiezza così grande, è possibile che le chiavi derivate, anche grazie all'utilizzo di una curva space-filling che preservi la località dei dati, tendano a formare dei cluster e renda inutile la divisione dello spazio in un numero elevato di intervalli in quanto la maggior parte delle chiavi saranno contenute in pochi intervalli. Per risolvere tale problema, in *HASP* si utilizza la soluzione basata sul *Data Driven BitVector*.

5.2.4 Strutture di digest: implementazione del Q-Digest in HASP

XCone implementa il *q-digest* mediante l'utilizzo di un vettore con un numero di elementi pari al numero di tutti i nodi dell'albero di *q-digest* e ne setta un flag opportuno nel caso in cui a quel nodo sia effettivamente associato un intervallo significativo. In memoria è mantenuta dunque una struttura dati che mantiene informazioni su tutti i nodi dell'albero di digest che possiamo indicare come q-digest *statico* dal momento che la realizzazione della struttura dati non è guidata dalla presenza dei dati reali nell'albero di *q-digest*. L'utilizzo di una strategia simile in *HASP* risulterebbe impraticabile, in quanto si otterrebbero facilmente vettori di dimensioni enormi non allocabili in memoria o difficilmente gestibili in un tempo ragionevole ed in maniera efficiente. Ad esempio, utilizzando sei attributi ($n = 6$) con 10 bit per contenere la rappresentazione binaria del valore di ciascun attributo ($k = 10$), sarebbe necessario allocare in memoria un vettore di $2^{60} - 1$ elementi soltanto per memorizzare i nodi foglia dell'albero. Dovendo mantenere informazioni su un numero così grande di chiavi derivate, non è possibile pensare di utilizzare il *q-digest statico* e si rivela pertanto necessaria la definizione di un *q-digest dinamico* che mantenga in memoria soltanto quegli elementi dell'albero di *q-digest* che forniscono informazioni significative.

Ricordiamo che su tale struttura dati è richiesto che sia possibile calcolare la proprietà di *q-digest* in maniera efficiente:

$$\text{count}(v) \; + \; \text{count}\,(v_p) \; + \text{count}(v_s) > \left\lfloor \frac{n}{k} \right\rfloor$$

Per rappresentare l'albero di *q-digest* è stata definita una nuova struttura dati *QDigestTree* definita mediante una *HashMap* indicizzata mediante i livelli dell'albero che includono almeno un nodo significativo dell'albero del *q-digest* e che associa ad ogni livello una lista ordinata di elementi (nodi).

Ciascun nodo dell'albero di *q-digest*, chiamato *QDigestElement*, è composto da tre campi principali:

- un campo *low*,
- un campo *up,*

- un contatore di risorse (*count*).

I campi *low* ed *up* indicano l'intervallo [*low; up*] a cui corrisponde tale nodo nell'albero. Le foglie dell'albero possiedono valori uguali nei campi *low* ed *up*. Per ciascun livello dell'albero è mantenuta una lista di elementi (nodi) appartenenti a tale livello, chiamata *QDigestLevel* e mantenuta ordinata secondo il valore del campo *low* degli elementi. In figura 5.10 è mostrato un esempio della rappresentazione di un albero di *q-digest*.

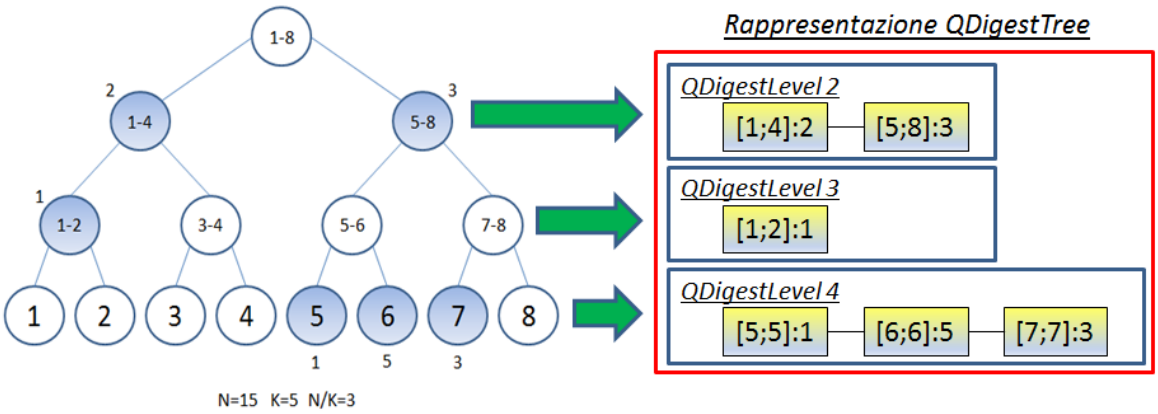


Figura 5.10 – Esempio di rappresentazione dell'albero di *q-digest*

Per stabilire se due nodi A e B, contenuti entrambi nella lista degli elementi del livello i-esimo sono fratelli, è necessario che siano verificate le seguenti due condizioni:

1. devono essere consecutivi, ovvero che $low_B = up_A + 1$
2. la loro unione $[\text{low}_A; \text{up}_B]$ deve essere un intervallo (nodo) valido al livello *i-1*-esimo dell'albero

Per stabilire se due nodi A e B sono consecutivi o meno, è necessario, ma non sufficiente, che sia verificata l'equazione

$$low_B = up_A + 1$$

Se tale condizione non è verificata, sicuramente i due nodi non sono consecutivi nell'albero (non sono fratelli), pur potendo essere consecutivi nella lista degli elementi appartenenti ad un determinato livello. Al contrario, nel caso in cui l'equazione

precedente sia verificata, per poter affermare che i due nodi sono sicuramente fratelli e dunque procedere al controllo combinato della proprietà di *q-digest*, è necessario controllare che la loro unione dia luogo ad un nodo valido per il livello inferiore. L'unione di due nodi A e B è data dal nodo di intervallo $[low_A; up_B]$ con count pari a $count_A + count_B$. Supponendo che i nodi A e B appartengano alla lista degli elementi del livello i+1-esimo, la loro unione, se valida, apparterrebbe al livello i-esimo. Conoscendo il massimo numero di foglie memorizzabili, σ, e dunque il numero massimo di livelli T_{liv}, che definiscono l'albero nella sua forma completa, in cui

$$T_{liv} = (log_2\, \sigma) + 1$$

è possibile dedurre, per ciascun livello i dell'albero, l'ampiezza degli intervalli che descrivono i nodi di quel determinato livello come

$$\gamma_i = 2^{T_{\text{liv}} - i} \text{ con } i = 1, ..., (log_2\, \sigma) + 1$$

A questo punto, noto il valore dell'ampiezza degli intervalli dei nodi del livello i-esimo, γ_i, è possibile determinare se il nodo dato dall'unione dei due nodi A e B è un nodo valido di livello i semplicemente effettuando un controllo sui valori low_A e up_B. Se al livello i-esimo esiste un nodo P tale per cui siano verificate entrambe le seguenti condizioni:

$$low_P = low_A \qquad (1)$$
$$up_P = low_P + \gamma_i = up_B \qquad (2)$$

allora è possibile affermare con certezza che i due nodi A e B di livello *i+1*-esimo sono fratelli e il nodo P è il loro padre di livello i-esimo.

Conoscendo l'ampiezza γ_i degli intervalli di livello i-esimo, si può verificare la proprietà (1) tramite l'utilizzo della seguente equazione:

$$\gamma_i \times H = low_A$$

A seconda del valore assunto dalla variabile H, è possibile stabilire se al livello i-esimo esiste un nodo P, padre dei due nodi A e B di livello *i+1*-esimo. In particolare se H assume un valore intero:

$$H \in I \Rightarrow \text{i nodi } A \text{ e } B \text{ sono fratelli \& il loro padre è}$$

$$[\text{low}_P = H \times \gamma_i; \text{up}_P = \text{low}_P + \gamma_i]$$

altrimenti *se* $H \notin I \Rightarrow$ i nodi A *e* B NON sono fratelli

Compress. Al fine di verificare e mantenere la proprietà di *q-digest* su ciascun nodo dell'albero, si esegue un algoritmo di compressione su ciascun livello dell'albero a partire dall'ultimo livello, il livello delle foglie, fino al livello 2, ovvero al livello immediatamente inferiore la radice (considerata il livello numero 1). La procedura *compress* chiamata sull'*i-esimo* livello dell'albero, verifica se:

1. il livello i esiste (un livello esiste se è presente almeno un nodo al suo interno)
2. per ogni nodo appartenente al livello i-esimo, controlla se la proprietà di *q-digest* è verificata o meno. Se non lo è, tale nodo (e suo fratello se esistente), vengono rimossi dal livello i-esimo e il *count* del nodo padre al livello *i-1*-esimo aumentato con la somma dei loro valori. Se il nodo padre al livello *i-1*-esimo non esiste, viene creato ed il suo *count* settato con la somma dei count dei nodi figli o pari al valore del *count* del nodo figlio se il fratello non esiste.

La struttura dati dinamica utilizzata per la memorizzazione dell'albero di *q-digest*, memorizza effettivamente solamente i nodi presenti nell'albero ed non tiene traccia di tutti i nodi, tra cui anche quelli non presenti come avviene nell'implementazione di *XCone*. I livelli dell'albero in cui non è presente alcun nodo, magari in seguito ad un'operazione *compress*, vengono rimossi e vengono tenuti in memoria solamente i livelli dell'albero con almeno un nodo presente.

$$\forall \ \textit{livello } i, \ i = (log_2\sigma) + 1, ..., 2 \ \text{tale che} \ |\text{livello}_i| \neq 0 \Rightarrow \textit{compress} \ (\textit{livello}_i)$$

Il seguente teorema 5.2.1 fornisce un limite superiore al numero di elementi memorizzati in un albero di *q-digest*.

Teorema 5.2.1. *La massima dimensione, o numero di nodi, di ciascun albero di* q-digest Q *è al massimo pari a* 3k*, dove k esprime il fattore di compressione dell'albero.*

Dimostrazione. Poichè tutti i nodi appartenenti all'albero di *q-digest* soddisfano la proprietà

$$\text{count}(v) + \text{count}(v_p) + \text{count}(v_s) > \left\lfloor \frac{n}{k} \right\rfloor$$

vale la seguente disuguaglianza:

$$\sum_{v \in Q} (\text{count}(v) + \text{count}(v_p) + \text{count}(v_s)) > |Q| \times \left\lfloor \frac{n}{k} \right\rfloor$$

in cui $|Q|$ rappresenta la dimensione dell'albero di *q-digest* Q. Adesso, considerando il membro sinistro della disequazione, il count di ciascun nodo contribuisce al massimo una volta come nodo padre, fratello o per se stesso. Quindi è possibile scrivere:

$$\sum_{v \in Q} (\text{count}(v) + \text{count}(v_p) + \text{count}(v_s)) \leqslant 3 \times \sum_{v \in Q} (\text{count}(v) = 3n$$

e da tale disequazione possiamo arrivare a

$$|Q| \times \frac{n}{k} < 3n$$

E' dunque possibile affermare che $3k$ è il limite superiore alla dimensione totale dell'albero di *q-digest*. □

Il risultato precedente consente di modellare la dimensione del *q-digest* variando opportunamente il valore del parametro k.

Esempio 5.2.2. Supponiamo ad esempio di avere un albero come quello in figura 5.11(A) e che il fattore di compressione k abbia un valore pari a 7. Nella lista degli elementi del livello delle foglie, in questo esempio è il livello 4, sono presenti cinque nodi. Il primo nodo di cui viene verificata la proprietà di *q-digest* è il nodo 2. Il nodo successivo nella lista al nodo 2 è, in questo esempio, il nodo 3, ma i due nodi pur essendo consecutivi non sono fratelli in quanto la loro unione [2;3] non dà come risultato un nodo valido di livello 3. Pertanto il nodo 2 viene considerato da solo e ne viene controllata la proprietà di *q-digest* assumendo che abbia un fratello con un *count* pari a zero. Poichè il nodo numero 2 ha un *count* pari a 3, superiore al valore soglia fissato $\left\lfloor \frac{n}{k} \right\rfloor = 2$, il nodo 2 non viene modificato. Si esamina quindi il nodo 3 di livello 4. Il nodo successivo nella lista del livello 4, il nodo numero 4, è consecutivo al nodo 3 e quindi è necessario controllare che la loro unione [3;4] sia un nodo valido di livello 3. La loro unione è un nodo valido (esiste il nodo [3;4] al livello 3) e dunque si controlla la proprietà di *q-digest* assumendo il nodo numero 4, fratello del nodo numero 3. La somma dei *count* dei due nodi (il padre [3;4] non esiste e quindi ha *count* pari a zero) non è maggiore stretta del valore soglia $\left\lfloor \frac{n}{k} \right\rfloor = 2$, pertanto i due nodi vengono rimossi

dal livello 4 e il nodo padre [3;4] aggiunto al livello 3 con un *count* pari a due. E' possibile vedere il risultato al termine di tale passo dell'algoritmo in figura 5.11(B).

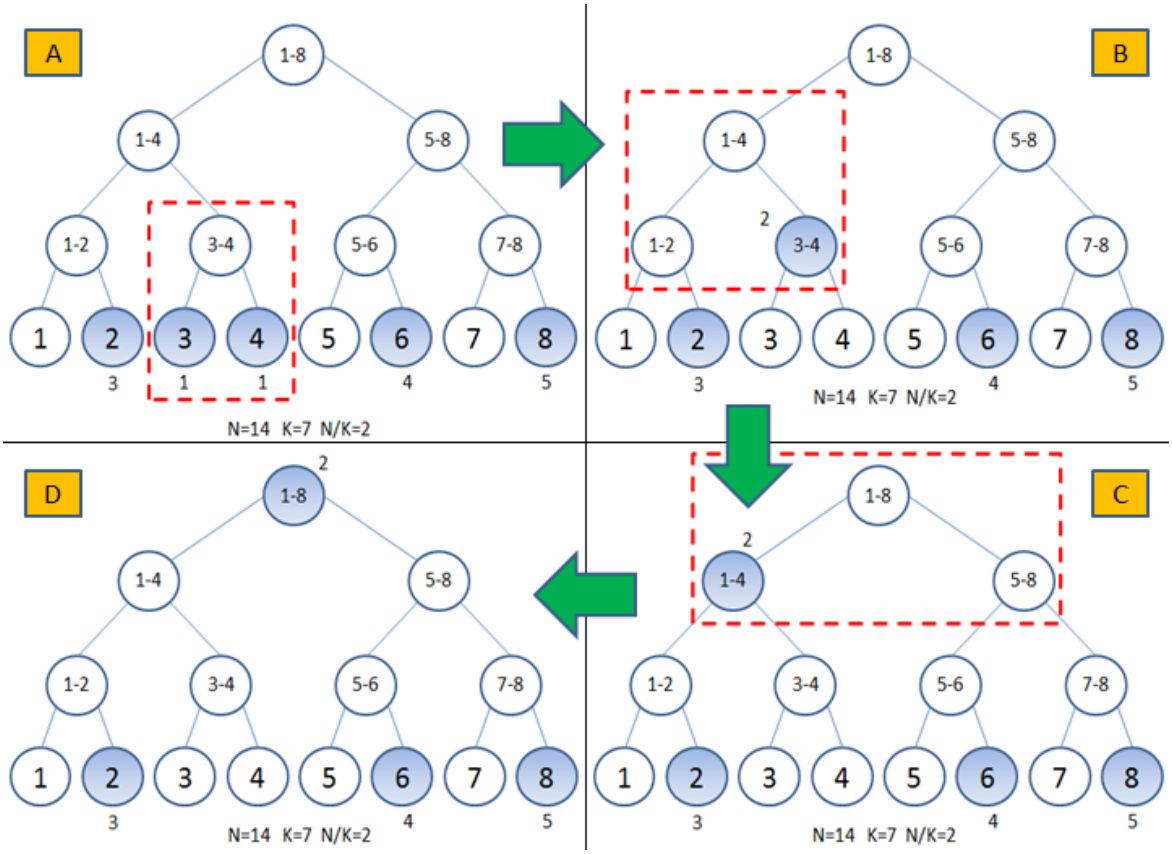


Figura 5.11 – Esempio di esecuzione dell'algoritmo di compressione dell'albero di *q-digest* con i parametri n=14, k=7 (Valore soglia $\frac{n}{k} = 2$

L'algoritmo controlla successivamente i nodi 6 ed 8 nella lista del livello 4, ma poichè essi verificano la proprietà di *q-digest* non subiscono alcuna modifica. L'operazione *compress* viene chiamata successivamente sul livello 3 dell'albero e si ricava che il nodo [3;4] non verifica la proprietà di *q-digest* e quindi viene rimosso ed il nodo padre [1;4] aggiunto al livello 2. (Figura 5.11(C)). Poichè il nodo [3;4] era l'unico nodo di livello 3, l'operazione *compress* sul tale livello termina e viene chiamata sul livello 2. Il nodo [1;4] non verifica la proprietà di *q-digest* e pertanto viene rimosso dal livello 2 e il nodo padre [1;8] aggiunto al livello 1. (Figura 5.11(D)). Il nodo [1;4] era l'unico

nodo di livello 2 e pertanto l'operazione *compress* su tale livello termina. Poichè l'ultimo livello esaminato, il livello 2, è il livello precedente la radice, l'algoritmo di compressione dell'albero di q-digest termina.

Merge. Questa funzione viene usata per unire due alberi di *q-digest* in un unico albero. Si crea un nuovo albero di *q-digest* in cui si aggiungono al livello i-esimo tutti i nodi presenti allo stesso livello i-esimo nei due alberi, applicando la seguente regola:

- se ad uno stesso livello i-esimo, un nodo è presente in entrambi gli alberi, si aggiunge il nodo al livello i-esimo nel nuovo albero una sola volta con un *count* pari alla somma dei *count* del nodo nei due diversi alberi di *q-digest* da unire.

- se un nodo è presente solamente in uno dei due alberi di *q-digest* al livello i-esimo, questo viene aggiunto senza modifiche nel nuovo albero al livello i-esimo

Al termine dell'aggiunta di tutti i nodi dei due alberi nel nuovo albero secondo la regola appena presentata, è necessario eseguire l'algoritmo di compressione su ogni livello del nuovo albero, sempre a partire dal livello delle foglie a salire fino al livello precedente la radice, in modo da ristrutturare l'albero affinchè ogni nodo rispetti la proprietà di *q-digest.* L'esecuzione dell'algoritmo *compress* sul nuovo albero è necessaria poichè, anche se i nodi dei singoli alberi rispettavano la proprietà di *q-digest*, è possibile che nel nuovo albero siano presenti uno o più nodi che la violano. Tale ristrutturazione dell'albero di *q-digest* inoltre comporta il ridimensionamento del numero di nodi dell'albero che viene mantenuto ad un valore sempre inferiore a $3k$, dove k esprime fattore di compressione dell'albero.

Esempio 5.2.3. Un esempio dell'unione di due alberi di *q-digest* è riportato in figura 5.12. Al termine, il nuovo albero di *q-digest* conterrà un numero di risorse n pari alla somma delle risorse n_1 ed n_2 memorizzate nei due singoli alberi; ciò modificherà il valore della variabile soglia $\lfloor \frac{n}{k} \rfloor$ per la aggregazione di nodi, in quanto il valore del fattore di compressione k rimane costante durante il processo di merge dei due alberi.

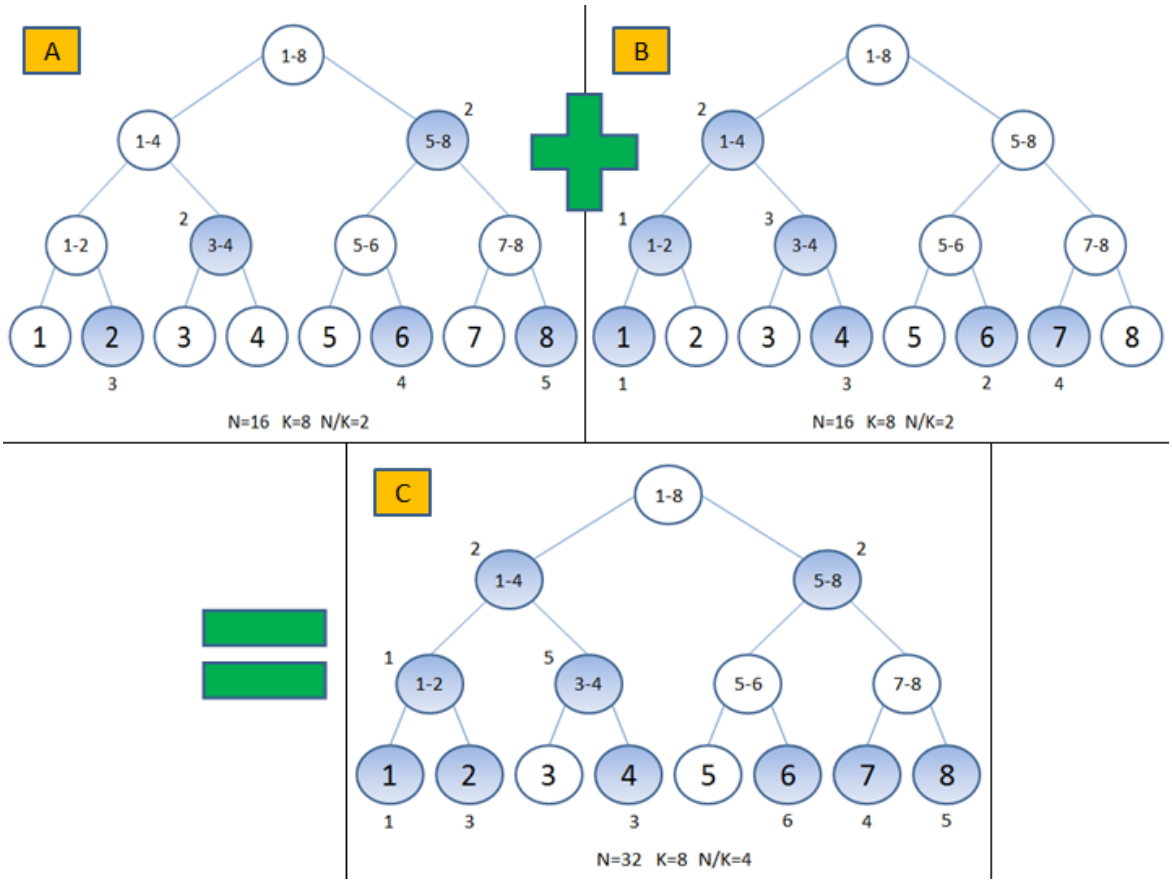


Figura 5.12 – Esempio di fusione di due alberi di *q-digest*

Capitolo 6

Risoluzione di query complesse

In questo capitolo viene affrontato il problema di come determinare le sezioni delle curve di Hilbert e Z-order che soddisfino una range query multidimensionale e dunque stabilire quali chiavi derivate soddisfano i vincoli imposti dalla query. Il problema principale per la risoluzione di query multidimensionali su curve space-filling è il passaggio dallo spazio multidimensionale dell'iper-rettangolo definito dalla range query allo spazio lineare definito sulla curva. Come vedremo in questo capitolo, gli approcci che possono essere seguiti sono di due tipi. Un approccio statico è quello di determinare tutti i segmenti di curva che sono contenuti nell'iper-rettangolo definito dalla range query senza considerare la distribuzione dei dati sulla curva space filling. Il problema principale di questo approccio è l'elevato numero di segmenti che vengono generati al crescere del numero delle dimensioni e del range di valori che gli attributi possono assumere. Un secondo approccio è quello *guidato dai dati*. Ricordiamo che il nostro scopo è quello di definire un algoritmo di risoluzione per query multidimensionali da integrare in *XCone*. Ricordiamo che l'albero di *HASP* è costruito a partire dalle chiavi derivate definite dagli attributi dei nodi e che i nodi interni contengono un *digest* che riassume mediante segmenti della curva space filling le chiavi contenute nel sotto-albero. Il secondo approccio può quindi essere realizzato secondo due diverse strategie. Una prima strategia consiste nel partire dalla query, individuare gli iper-quadranti che la intersecano e raffinare solo gli iper-quadranti che contengono dei dati. Il controllo che un iper-quadrante contenga dei dati può essere effettuato intersecando il segmento di curva corrispondente a quel quadrante con gli intervalli contenuti nel *digest*. Solo se l'intersezione tra il segmento di curva relativo ad un iper-quadrante che interseca la query ed il *digest* è diversa da zero, allora si procede al raffinamento.

Si evita così di generare segmenti di curva contenuti nella query, ma corrispondenti a zone della curva space filling non corrispondenti ad alcun dato. La seconda strategia parte invece dai segmenti contenuti nel *digest*, trova gli iper-quadranti che ricoprono tali segmenti e ne effettua quindi l'intersezione con la range query. In questo caso, si hanno a disposizione un insieme di intervalli di curva in cui sicuramente sono presenti dei dati e si deve passare da tali intervalli agli iper-quadranti che li contengono per poi intersecare questi quadranti con l'iper-rettangolo che descrive la range query. Tutti questi approcci verranno discussi nelle sezioni seguenti.

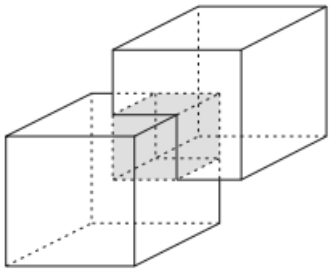


Figura 6.1 – Esempio di intersezione tra due iper-rettangoli in 3 dimensioni

6.1 La curva Z-order: intersezione con un iper-rettangolo

L'approccio statico determina tutti i segmenti della curva *Z-order* che sono contenuti nell'iper-rettangolo definito dalla range query a prescindere dai dati contenuti nel digest. Il problema principale di questo approccio è l'elevato numero di segmenti che vengono generati al crescere del numero delle dimensioni (n) e del range di valori che gli attributi possono assumere (k). In figura 6.2 è mostrato un esempio di range query multiattributo sulla curva *Z-order* con $n=2$ e $k=3$. E' possibile notare come il numero di segmenti della *Z-curve* contenuti nell'iper-rettangolo definito dalla range query sia già piuttosto elevato per valori molto bassi di n e di k.

Nell'esempio in figura 6.2 vengono generati 7 segmenti. Si noti che in generale il numero di segmenti è più elevato se si considerano query in cui i lati del rettangolo hanno dimensioni molto diverse. Con l'aumentare delle dimensioni della curva e del

range di valori assunto da ciascun attributo, il numero di segmenti di *Z-curve* contenuti nell'iper-rettangolo descritto dalla range query esplode.

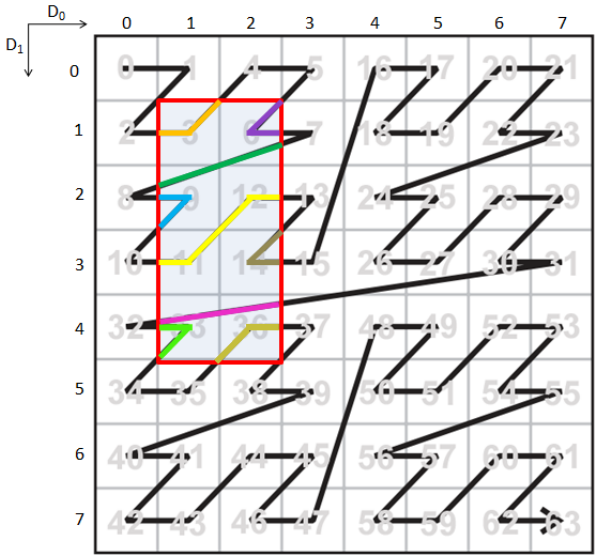


Figura 6.2 – Esempio dei segmenti di curva *Z-order* (2D) contenuti nell'iper-rettangolo definito dalla range query (rettangolo rosso).

Una volta ricavati tutti i segmenti della curva space filling che soddisfano la range query, non è detto che tutte le chiavi derivate appartenenti ai segmenti calcolati siano effettivamente memorizzate nei nodi *HASP*. Così facendo, è possibile che soltanto una piccola percentuale dei segmenti calcolati contenga informazioni significative. Un secondo approccio possibile per la risoluzione della range query multidimensionale è quello guidato dai dati che che verrà analizzato nella sezione successiva.

6.2 La curva Z-order: soluzione guidata dai dati

In questo approccio si suppone di essere a conoscenza della presenza di chiavi derivate in un intervallo $[\alpha\ ;\ \beta]$. Come abbiamo visto nel capitolo 5, gli intervalli di chiavi derivate corrispondenti a dati significativi saranno memorizzati nei nodi interni dell'albero *HASP,* secondo diverse tecniche di digest quali *BitVector* e *q-digest.*

Per poter calcolare se le chiavi appartenenti all'intervallo $[\alpha\ ;\ \beta]$ soddisfano la range query multidimensionale specificata, è necessario prima determinare la *Z-region envelope* ovvero gli iper-quadranti, che ricoprono la regione di curva compresa tra

due chiavi derivate α e β e successivamente effettuare l'intersezione tra questi iper-quadranti e l'iper-rettangolo definito dalla range query multidimensionale.

Il processo che permette di determinare quali parti della *curva Z-order* soddisfano la query multidimensionale comprende tre passi:

- determinazione della *Z-region envelope* compresa tra due chiavi derivate a partire dalla rappresentazione binaria degli iper-quadranti restituiti dal primo algoritmo
- calcolo per ciascuno delle coordinate geometriche
- test di intersezione degli iperquadranti determinati al passo precedente con la range query multidimensionale.

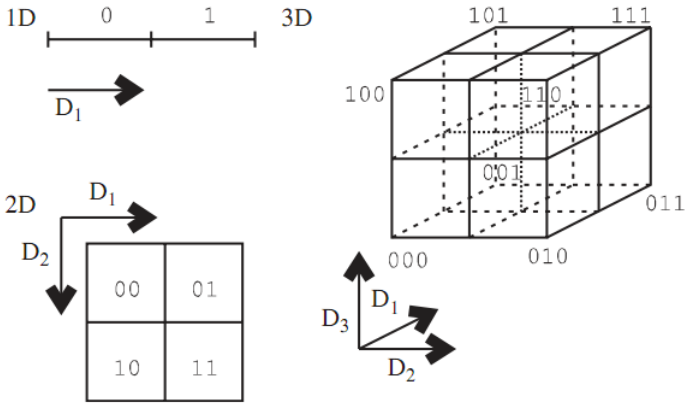


Figura 6.3 – Iper-quadranti e i loro codici

Tutti gli algoritmi descritti di seguito e che permettono di determinare se un segmento appartenente alla *curva Z-order* soddisfa la range query multidimensionale, si basano sull'analisi degli identificatori di ciascun iper-quadrante chiamati *navigation-codes*. Per poter dare la definizione di *navigation code* di un iper-quadrante è necessario prima definire il significato di *hquad code* di un iper-quadrante.

Definizione 6.2.1. *L'hquad code (hyperquadrant code) è una stringa binaria in cui ogni bit rappresenta la locazione dell'iper-quadrante in una certa dimensione. In particolare l'i-esimo bit viene impostato al valore 0 se l'hquad è posizionato nella prima metà di quella dimensione o al valore 1 se posizionato nella seconda metà.*

La definizione di *navigation code* di un iper-quadrante è la seguente:

Definizione 6.2.2. *Sia dato un iper-quadrante HQ di livello l. La stringa binaria* $navi(HQ) = c_0, c_1, ..., c_{l-1}$ *è chiamata navigation code dell'iper-quadrante HQ se ciascuna sottostringa* c_i *rappresenta il codice dell'iper-quadrante di livello i-esimo che contiene, dal punto di vista spaziale, l'iper-quadrante HQ.*

E' possibile effettuare una suddivisione ricorsiva dello spazio n-dimensionale in modo che ad ogni passo di suddivisione dello spazio corrispondano 2^n quadranti. In questo modo è possibile associare ad un iper-quadrante q di livello l un codice che identifica, ai diversi livelli della suddivisione, gli *hquad* che contengono q.

Da tale definizione è possibile derivare la seguente osservazione: *maggiore è* la dimensione *di un iper-quadrante, minore è la dimensione del suo navigation code e viceversa.*

In figura 6.3 sono mostrati gli *hquad codes* dei vari iper-quadranti nel caso unidimensionale, bidimensionale e tridimensionale, mentre in figura 6.4 è mostrato un esempio del processo di costruzione del *navigation code* di un iper-quadrante.

La proprietà che permette di ricavare un algoritmo per la risoluzione di range query multidimensionali sulla curva *Z-order* rispetto alla curva di Hilbert è espressa dal seguente teorema:

Teorema 6.2.3. *Il navigation code* $navi(o)$ *con* $o \in \Omega$ *è equivalente alla chiave derivata di o sulla curva Z-order,* $Z_{addr}(o)$*. Dunque* $navi(o) = Z_{addr}(o)$.

Tale teorema non è assolutamente valido per la curva di Hilbert, in cui la chiave derivata non è in relazione con l'identificatore dell'iper-quadrante a cui appartiene. Tale proprietà della curva *Z-order* è la chiave che ha permesso di poter ricavare gli algoritmi necessari per la risoluzione di range query multidimensionali tramite la linearizzazione dello spazio multidimensionale data da questa curva space-filling.

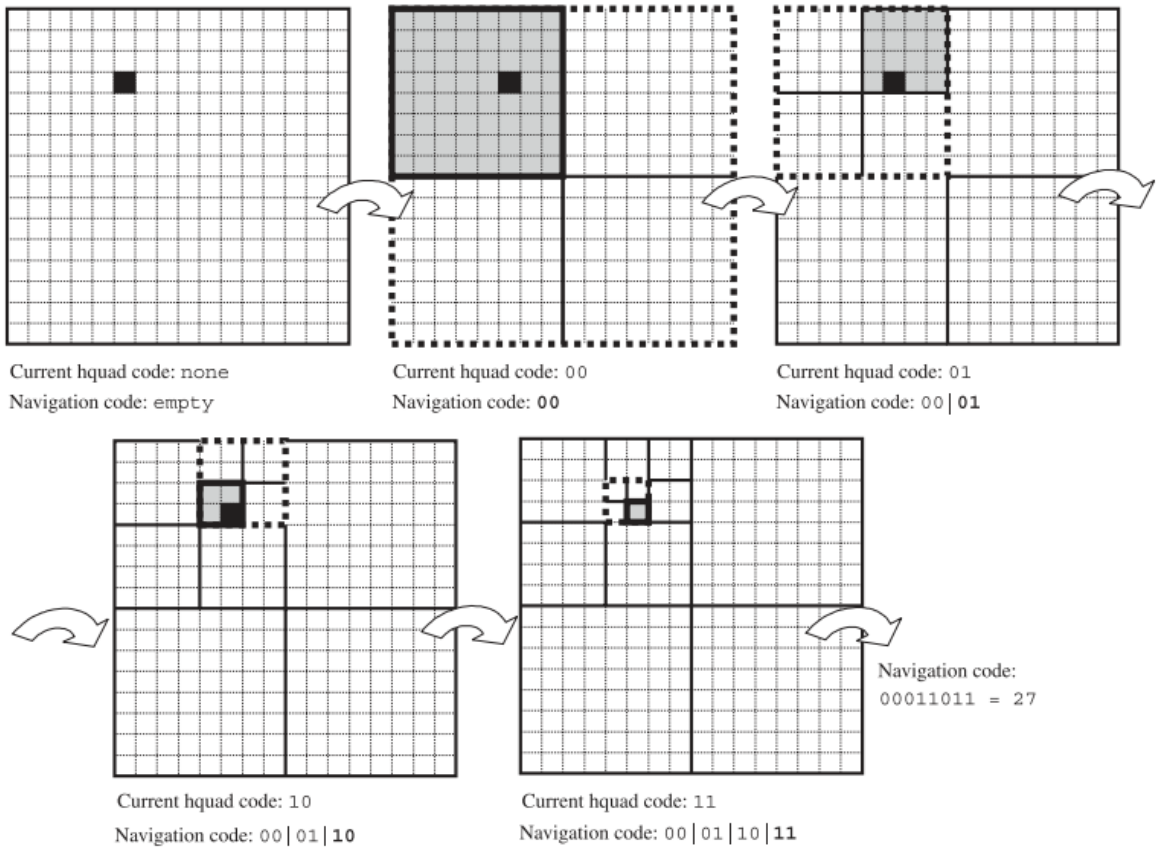


Figura 6.4 – Processo di costruzione del navigation code dell'iper-quadrante determinato dal punto di coordinate (6;2)

6.2.1 Z-region envelope tra due chiavi derivate

La regione della *Z-curve*, compresa tra due chiavi derivate α e β, ad esempio ricavate dal vettore di digest in *HASP* o con la strategia di aggregrazione *BitVector* o di *q-digest,* è calcolata utilizzando direttamente la rappresentazione binaria del valore delle due chiavi derivate ed esaminando gruppi di n bits (n pari al numero di dimensioni) per volta a partire dal bit più significativo delle due chiavi.

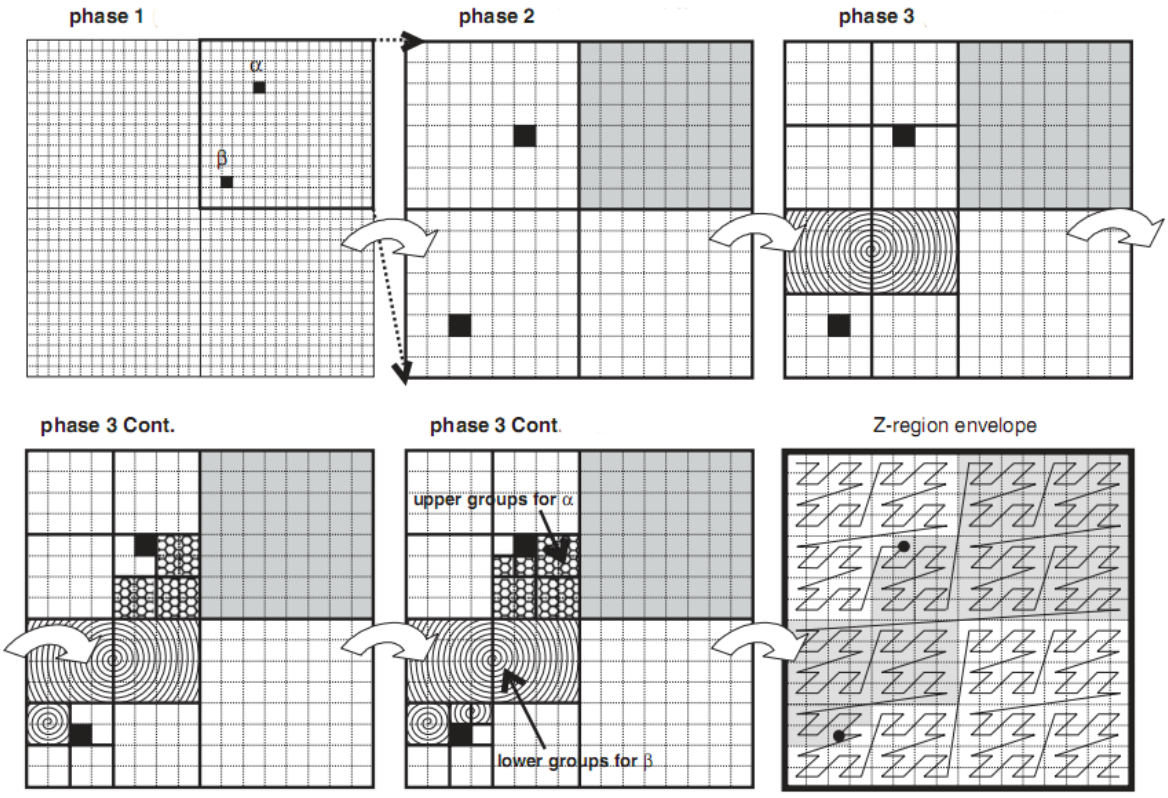


Figura 6.5 – Rappresentazione grafica delle iterazioni seguite dall'algoritmo di generazione della *Z-region envelope* compresa tra due chiavi derivate α e β

L'algoritmo, ispirato a [24], si compone di tre passi principali:

1. il primo passo considera la rappresentazione binaria di α e di β e calcola, se esiste, un quadrante o iper-quadrante minimo, che contenga entrambe le chiavi derivate. Ciò equivale a determinare se esiste un prefisso comune di n bits o di un multiplo di n bits tra le due chiavi derivate, con n pari al numero di dimensioni della curva. Tutti i quadranti, escluso quello individuato, possono essere trascurati nella fase successiva dell'algoritmo perchè sicuramente non conterranno alcuna chiave derivata del segmento $[\alpha\ ;\ \beta]$.

2. nel secondo passo si inizializza un prefisso p corrispondente all'iper-quadrante minimo che contiene le due chiavi derivate α e β, se esiste, e si esaminano iterativamente i gruppi di n bits successivi della chiave derivata α e si aggiungono

alla lista degli iper-quadranti che compongono la *Z-region envelope*, gli iper-quadranti aventi codifica binaria pari al prefisso p corrente e i successivi n bits con valore maggiore degli n bits correntemente selezionati di α e, soltanto alla prima iterazione, con valore minore dei corrispondenti n bits di β successivi ai bit del prefisso comune tra le due chiavi. Al termine di ogni iterazione si calcola il nuovo valore di p concatendando al valore attuale di p gli n bits esaminati, tramite uno shift verso sinistra di n posizioni del prefisso corrente e una addizione del valore degli n bits esaminati. Questo procedimento si ripete finchè non sono stati esaminati tutti gli $n \times k$ bits della chiave derivata α e a quel punto si aggiunge la chiave derivata α alla lista degli iper-quadranti che compongono la *Z-region envelope*.

3. nella terza ed ultima parte dell'algoritmo si applica lo stesso procedimento alla chiave derivata β e si aggiungono alla lista degli iper-quadranti in output, gli iper-quadranti aventi codifica binaria pari al prefisso corrente e i successivi n bits con valore minore degli n bits correntemente selezionati di β e, solo alla prima iterazione, con valore minore degli n bits di α successivi ai bits del prefisso comune tra le due chiavi. Al termine dell'iterazione si concatenano al prefisso corrente gli n bits esaminati e si ripete il procedimento, finchè non sono stati esaminati tutti gli $n \times k$ bits della chiave derivata β. Nell'esame della chiave derivata β si può partire ad esaminare direttamente i bits in posizione $2n$ rispetto a quelli del prefisso comune tra le due chiavi in quanto l'iper-quadrante che viene calcolato alla prima iterazione (primi n bit diversi nelle rappresentazioni binarie delle due chiavi derivate) è già stato calcolato al passo precedente. Al termine dell'esame di tutti gli $n \times k$ bits che compongono la chiave derivata β, si aggiunge quest'ultima alla lista degli iper-quadranti che compongono la *Z-region envelope*.

In figura 6.5, la parte indicata come *phase1* corrisponde al primo passo dell'algoritmo in cui viene individuato il quadrante che contiene le chiavi α e β, estremi dell'intervallo. I quadranti rimanenti vengono trascurati nelle fasi successive dell'algoritmo. Nelle successive parti in figura 6.5 sono mostrati gli iper-quadranti individuati dai due passi successivi dell'algoritmo.

Esempio 6.2.4. Supponiamo di avere due chiavi derivate di valore $\alpha = 18$ e $\beta = 30$. La loro rappresentazione binaria sarà rispettivamente 010010 e 011110 con $n = 2$ ed

un ordine della curva $k = 3$. Il minimo iper-quadrante ($\mathrm{hq}_{\min}$) che contiene entrambe le chiavi derivate è dato dal prefisso comune di 2 (o un multiplo di 2) bits a partire dal bit più significativo ed in questo caso è pari a 01.

$$\alpha = 18 \ = 010010_2$$

$$\beta = 30 \ = 011110_2$$

$$\mathrm{hq}_{\min} = 01$$

La ricerca degli iper-quadranti proseguirà quindi all'interno dell'iper-quadrante con *navigation code* pari a 01.

Esempio 6.2.5. Il prefisso comune tra le due chiavi derivate dell'esempio 6.2.4 precedente è 01. Alla prima iterazione nell'esame della chiave derivata α, i successivi n bit (con $n = 2$) hanno valore 00, mentre quelli di β valore 11. Si aggiungono alla *Z-region envelope* tutti gli iper-quadranti con codifica binaria formata dal prefisso comune e da un valore dei 2 bits successivi maggiore di 00 e minore di 11, estremi non compresi: 0101 e 0110. A questo punto si aggiungono gli n bits della chiave α correntemente selezionati al prefisso corrente che diventa pari a 0100 e si esegue nuovamente lo stesso procedimento, esaminando i successivi n bits di α, senza considerare stavolta il corrispettivo valore degli n bits di β . Gli iper-quadranti che verranno aggiunti in questa iterazione, sono lo 010011 ed essendo l'ultima iterazione anche la chiave derivata $\alpha = 010010$ viene inserita nella *Z-region envelope.*

Il numero di iterazioni del ciclo principale dell'algoritmo, nel caso peggiore, è pari all'ordine della curva k, in quanto per l'analisi degli $n \times k$ bits che compongono le chiavi derivate α e β, si esaminano gruppi di n bits per volta. Il caso peggiore è il caso in cui le due chiavi derivate non possiedono alcun prefisso in comune e l'iper-quadrante minimo che le contiene entrambe è l'iper-quadrante di dimensione massima; se invece le due chiave possiedono n o un multiplo di n bits in comune, il numero di iterazioni eseguite, nel ciclo principale dell'algoritmo, sarà tanto minore quanto maggiore sarà la lunghezza del prefisso comune.

Mostriamo infine un esempio completo di generazione della *Z-region envelope.*

Esempio 6.2.6. Consideriamo il caso bidimensionale ($n = 2$) e con attributi che possono assumere un valore compreso nell'intervallo tra [0 ; 7], quindi con un ordine

della curva $k = 3$. Vogliamo ricavare la regione di Z-curve compresa tra le chiavi derivate $\alpha = 13$ e $\beta = 55$. Il primo passo da seguire è la conversione del valore delle due chiavi derivate nella loro rappresentazione binaria su $n \times k$ bits.

$$\alpha = 13 = 001101_2$$

$$\beta = 55 = 110111_2$$

Dall'esame dei bits delle due chiavi derivate, a partire dal bit più significativo, si deduce che non c'è un iper-quadrante di grandezza minore rispetto all'iper-quadrante di dimensione massima in cui sono contenute entrambe le chiavi derivate α e β. L'algoritmo parte dunque dall'esame dei primi $n = 2$ bits delle due chiavi derivate. I primi due bit di α sono pari a 00 mentre quelli di β pari a 11; poichè non esiste un prefisso comune tra le due chiavi, il prefisso corrente *currentPrefix* è inizializzato alla stringa vuota e gli iper-quadranti che sono aggiunti alla lista di output sono quelli ottenuti concatenando tale stringa con le stringhe binarie comprese nell'intervallo (00 ; 11) e quindi:

$$\mathrm{hq}_0 = 01$$

$$\mathrm{hq}_1 = 10$$

Al termine delle prima iterazione si aggiorna il valore del prefisso corrente aggiungendo il valore dei successivi n bits della chiave α. Il prefisso corrente diventa dunque pari a $\mathrm{current}_{\mathrm{prefix}} = 00$. Per semplicità e maggiore chiarezza nell'esposizione, l'esame dei bits delle chiavi derivate α e β saranno qui riportati uno di seguito all'altro, mentre durante l'algoritmo sono essere eseguiti contemporaneamente.

Nella seconda iterazione si esaminano i successivi $n = 2$ bits di α, quindi il terzo e quarto bit della rappresentazione della chiave α, che hanno valore 11. Si aggiungono alla lista di output tutti gli iper-quadranti formati dalla concatenazione di *currentPrefix* con gli ultimi $n = 2$ bits di valore maggiore dei bits di α correntemente in esame. Poichè il terzo e quarto bit della rappresentazione binaria del valore della chiave derivata α sono pari ad 11 e possiedono dunque già il valore massimo esprimibile su $n = 2$ bits, in questa iterazione non si aggiunge alcun iper-quadrante alla lista della *Z-region envelope*. Poichè il numero di bits esaminati ($2 \times n = 4$) è minore del numero totale

di bits ($n \times k = 6$) della rappresentazione binaria della chiave derivata α, si esegue un'altra iterazione e si aggiorna il prefisso corrente ($\text{current}_{\text{prefix}} = 00$) effettuando uno shift verso sinistra di $n = 2$ bits del valore attuale del prefisso corrente ($\text{current}_{\text{prefix}} = 0000$) e aggiungendovi il valore degli $n = 2$ bits in esame di α, ottenendo un valore per il prefisso corrente pari a $\text{current}_{\text{prefix}} = 0011$.

Nella terza ed ultima iterazione si esaminano gli $n = 2$ bits successivi della chiave α e quindi il quinto e sesto bit della rappresentazione binaria di α, che hanno valore 01. Si aggiungono alla lista di output tutti gli iper-quadranti formati da *currentPrefix* e dagli ultimi $n = 2$ bits di valore maggiore dei bits di α correntemente in esame. Poichè il quinto e sesto bit della rappresentazione binaria del valore della chiave derivata α sono pari a 01 ed il prefisso corrente ha valore $\text{current}_{\text{prefix}} = 0011$, si aggiungono alla lista di output i seguenti iper-quadranti:

$$\text{hq}_2 = 001110$$

$$\text{hq}_3 = 001111$$

Poichè il numero di bits di α esaminati ($3 \times n = 6$) è pari al umero totale di bits ($n \times k = 6$) della rappresentazione binaria della chiave derivata α, non si eseguono ulteriori iterazioni per quanto riguarda la chiave α e si aggiunge la chiave derivata α stessa alla lista degli iper-quadranti che compongono la *Z-region envelope*.

$$\text{hq}_4 = \alpha = 011101$$

A questo punto lo stesso procedimento va applicato alla chiave derivata $\beta = 110111_2$ e poichè il confronto tra i primi $n = 2$ bits diversi tra le rappresentazioni binarie delle due chiavi derivate α e β è stato effettuato nella prima iterazione dell'esame della chiave derivata α, si salta la prima iterazione sui bits della chiave β e l'unica operazione da effettuare è l'aggiornamento del prefisso corrente, che diventa pari ai primi $n = 2$ bits della chiave β ($\text{current}_{\text{prefix}} = 11$).

Nella seconda iterazione si esaminano i successivi $n = 2$ bits di β, quindi il terzo e quarto bit della rappresentazione della chiave β, che hanno valore 01. Si aggiungono alla lista di output tutti gli iper-quadranti formati da *currentPrefix* e dagli ultimi $n = 2$ bits con valore minore dei bits di β correntemente in esame. Poichè il terzo e quarto

bit della rappresentazione binaria del valore della chiave derivata β hanno valore pari a 01 e $\text{current}_{\text{prefix}} = 11$, si aggiungono alla lista di output i seguenti iper-quadranti:

$$\text{hq}_5 = 1100$$

Poichè il numero di bits esaminati ($2 \times n = 4$) è minore del numero totale di bits ($n \times k = 6$) della rappresentazione binaria della chiave derivata β, si esegue un'altra iterazione e si aggiorna il prefisso corrente ($\text{current}_{\text{prefix}} = 11$), effettuando uno shift verso sinistra di $n = 2$ bits del valore attuale($\text{current}_{\text{prefix}} = 1100$) e aggiungendovi il valore degli $n = 2$ bits in esame di β, ottenendo un valore per il prefisso corrente pari a $\text{current}_{\text{prefix}} = 1101$.

Nella terza iterazione si esaminano gli $n = 2$ bits successivi della chiave β e quindi il quinto e sesto bit della rappresentazione binaria di β, che hanno valore 11. Si aggiungono alla lista di output tutti gli iper-quadranti formati dal prefisso corrente e dagli ultimi $n = 2$ bits di valore minore dei bits di β correntemente in esame. Poichè il quinto e sesto bit della rappresentazione binaria del valore della chiave derivata β sono pari a 11 ed il prefisso corrente ha valore $\text{current}_{\text{prefix}} = 1101$, si aggiungono alla lista di output i seguenti iper-quadranti:

$$\text{hq}_6 = 110110$$

$$\text{hq}_7 = 110101$$

$$\text{hq}_8 = 110100$$

Poichè il numero di bits di β esaminati ($3 \times n = 6$) è pari al numero totale di bits ($n \times k = 6$) della rappresentazione binaria della chiave derivata β non si eseguono ulteriori iterazioni per quanto riguarda la chiave β e si aggiunge la chiave derivata β stessa alla lista degli iper-quadranti che compongono la *Z-region envelope*.

$$\text{hq}_9 = \beta = 110111$$

La *Z-region envelope* contenuta tra le due chiavi derivate $\alpha = 13$ e $\beta = 55$, al termine dell'algoritmo, è mostrata in figura 6.6, nella quale sono messi in evidenza, ciascuno con un colore diverso, gli iper-quadranti calcolati.

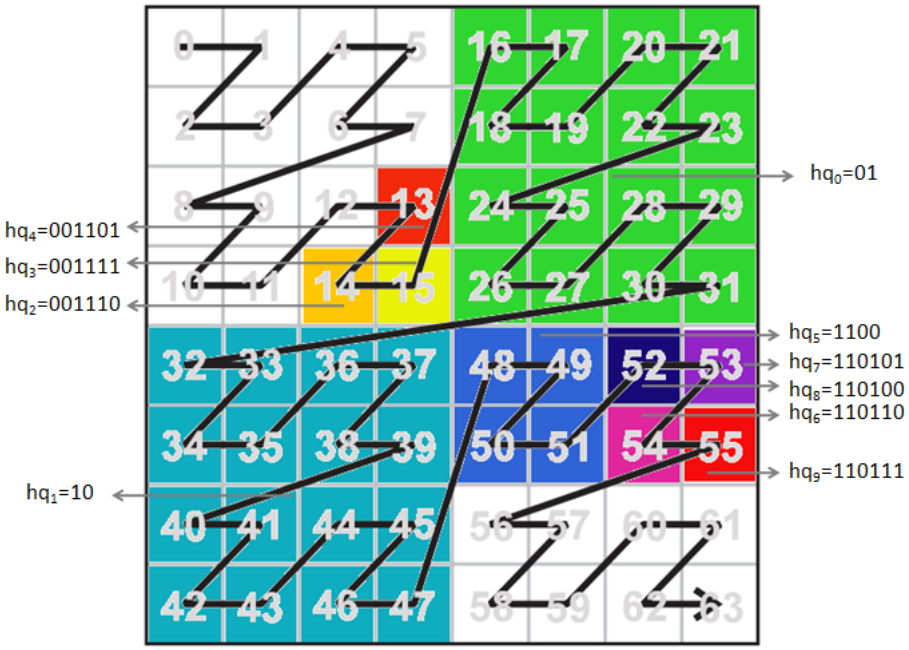


Figura 6.6 – Quadranti di output dell'intervallo di chiavi derivate [13;55]

La *z-region envelope* compresa tra le chiavi derivate $\alpha = 13$ e $\beta = 55$ è data dall'unione degli iper-quadranti contenuti nella lista di output.

$$\text{zRegionEnv}_{13-55} = \{\text{hq}_0, \text{hq}_1, ..., \text{hq}_9\}$$

L'intersezione, se non nulla, tra gli iper-quadranti che formano tale *z-region envelope* e la range query determinerà le chiavi derivate, e quindi le risorse, che soddisfano i vincoli imposti dalla range query multidimensionale.

6.2.2 Calcolo delle coordinate geometriche di un iper-quadrante

Al fine di determinare se un iper-quadrante è intersecato o meno dalla range query e quindi contiene chiavi derivate che soddisfano i vincoli della query, è necessario effettuare l'intersezione tra la query e l'iper-quadrante. Per fare ciò è necessario ricavare, a partire dal *navigation code* dell'iper-quadrante, le coordinate geometriche che lo caratterizzano. A tale scopo si esaminano uno alla volta tutti i bit della rappresentazione binaria dell'iper-quadrante a partire dal più significativo. A partire dal numero di dimensioni n e dall'ordine k della curva si ricavano gli n intervalli che definiscono

l'iper-quadrante come divisioni successive dell'intervallo iniziale $[0\,;(2^k-1)]$ che esprime i valori possibili di ogni attributo. Al passo zero dell'algoritmo si inizializzano dunque tutti gli n intervalli con l'intervallo iniziale e si esaminano i bit del *navigation code* dell'iper-quadrante: se l'i-esimo bit è pari a 0, si divide a metà il valore corrente dell'intervallo della dimensione $i \bmod n$ e si considera la parte sinistra dell'intervallo, mentre se l'i-esimo bit ha valore 1 si considera quella destra. Il primo passo dell'algoritmo prevede l'analisi del valore del bit più significativo, l'intervallo della prima dimensione è inizializzato a $[0\,;(2^k-1)]$ e nel caso in cui il primo bit abbia valore 0, l'intervallo sarà aggiornato al valore $[0\,;(2^{k-1}-1)]$, mentre se ha valore 1 al valore $[2^{k-1}\,;2^k-1)\,]$. Si procede in questo modo tramite divisioni successive degli intervalli delle n dimensioni fino a che non sono stati esaminati tutti i bits che compongono il *navigation code* dell'iper-quadrante in esame.

Esempio 6.2.7. Consideriamo il caso bidimensionale ($n = 2$) con attributi che possano assumere un valore compreso nell'intervallo [0 ; 15], quindi con un ordine della curva $k = 4$; si vogliono determinare le coordinate geometriche del quadrante $\mathrm{hq}_{\mathrm{input}}$ di valore 0110.

$$\mathrm{hq}_{\mathrm{input}} = 0110$$

Dobbiamo determinare gli $n = 2$ intervalli che definiscono tale quadrante. Al passo zero dell'algoritmo gli $n = 2$ intervalli sono inizializzati al valore indicato dall'ordine della curva: $[0\,;\,(2^k-1)]$ quindi per ciascuna delle n dimensioni, x_i, con $i = 1, 2$:

$$0 \leqslant x_i \leqslant 15$$

L'algoritmo esamina, uno alla volta, tutti i bits che compongono la codifica binaria del quadrante; alla prima iterazione dell'algoritmo, si esamina il bit in posizione zero (il bit più significativo), il quale ha valore 0. Il bit in posizione zero si riferisce alla prima dimensione. Poichè il valore di tale bit è zero, l'intervallo della prima dimensione viene diviso a metà e viene considerata la parte sinistra di questa divisione:

$$0 \leqslant x_0 \leqslant 7$$

Alla seconda iterazione, il bit in posizione uno ha valore 1, ma stavolta è riferito alla seconda dimensione e perciò l'intervallo di x_1 diventa:

$$8 \leqslant x_1 \leqslant 15$$

in quanto l'intervallo corrente è suddiviso a metà e ne viene considerata la parte destra.

Alla terza iterazione, il bit in posizione due ha valore 1 e l'intervallo della prima dimensione viene ulteriormente suddiviso e assume il valore:

$$4 \leqslant x_0 \leqslant 7$$

Alla quarta ed ultima iterazione, si esamina il bit in posizione tre, il quale possiede un valore pari a 0. L'intervallo della seconda dimensione viene suddiviso e ne viene considerata la parte sinistra:

$$8 \leqslant x_1 \leqslant 11$$

Al termine dell'algoritmo gli intervalli finali ottenuti descrivono l'iper-quadrante con *navigation code* $\mathrm{hq}_{\mathrm{input}} = 0110$ e sono pari a:

$$4 \leqslant x_0 \leqslant 7$$

$$8 \leqslant x_1 \leqslant 11$$

6.2.3 Intersezione tra un iper-quadrante e una range query

Al fine di poter stabilire se un iper-quadrante soddisfa o meno i vincoli di una range query è necessario determinare un criterio di intersezione. Una range query è definita da n intervalli che definiscono un iper-rettangolo. Il criterio di intersezione deve poter dunque indicare, a livello geometrico, se un iper-rettangolo interseca un iper-quadrante. Ricordiamo che ciascuna range query è definita da n intervalli del tipo:

$$\mathrm{low}^i_{\mathrm{query}} \leqslant x_i \leqslant \mathrm{high}^i_{\mathrm{query}}$$

dove $\mathrm{low}^i_{\mathrm{query}}$ e $\mathrm{high}^i_{\mathrm{query}}$ rappresentano il valore minimo e massimo dell'intervallo della *i-esima* dimensione della range query e che ciascun iper-quadrante è definito da n intervalli del tipo:

$$\mathrm{low}^i_{\mathrm{hq}} \leqslant x_i \leqslant \mathrm{high}^i_{\mathrm{hq}}$$

dove low^i_{hq} e $\text{high}^i_{\text{hq}}$ rappresentano il valore minimo e massimo dell'intervallo della *i-esima* dimensione dell'iper-quadrante.

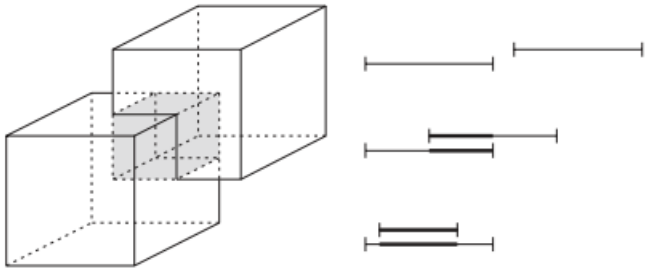


Figura 6.7 – Esempio di intersezione tra iper-rettangoli e tra gli intervalli che li descrivono

Una range query interseca un iper-quadrante se la seguente condizione vale per ciascuna delle n dimensioni:

$$|\text{low}^i_{\text{hq}} - \text{low}^i_{\text{query}}| + |\text{high}^i_{\text{hq}} - \text{high}^i_{\text{query}}| \leqslant |\text{low}^i_{\text{hq}} - \text{high}^i_{\text{hq}}| + |\text{low}^i_{\text{query}} - \text{high}^i_{\text{query}}|$$

ovvero se vi è intersezione, in tutte le dimensioni, tra gli intervalli che definiscono l'iper-rettangolo descritto dalla range query e l'iper-quadrante appartenente alla *Z-curve*.

I valori $|\text{low}^i_{\text{query}} - \text{high}^i_{\text{query}}|$ e $|\text{low}^i_{\text{hq}} - \text{high}^i_{\text{hq}}|$ rappresentano rispettivamente la lunghezza del lato della query e dell'iper-quadrante rispetto alla i-esima dimensione. Il valore $|\text{high}^i_{\text{hq}} - \text{high}^i_{\text{query}}|$ è uguale al lato della query e cioè a $|\text{low}^i_{\text{query}} - \text{high}^i_{\text{query}}|$ solo se i valori delle lunghezze dei due lati coincidono. Se tale valore è maggiore, sicuramente non vi è intersezione tra l'iper-quadrante e la range query (fig. 6.8(b)). Al contrario, se $|\text{high}^i_{\text{hq}} - \text{high}^i_{\text{query}}|$ è minore stretto della lunghezza del lato dell'iper-quadrante, sicuramente c'è intersezione tra questo e l'iper-rettangolo descritto dalla query rispetto alla i-esima dimensione (fig. 6.8(a)). Se la condizione di intersezione è verificata per ciascuna delle n dimensioni, è possibile affermare che c'è sicuramente intersezione tra l'iper-quadrante e l'iper-rettangolo descritto dalla range query multidimensionale. Se non c'è intersezione rispetto anche soltanto ad una sola delle n dimensioni, è possibile affermare con sicurezza che non vi è alcuna intersezione tra l'iper-quadrante appartenente alla curva *Z-order* e la range query multidimensionale.

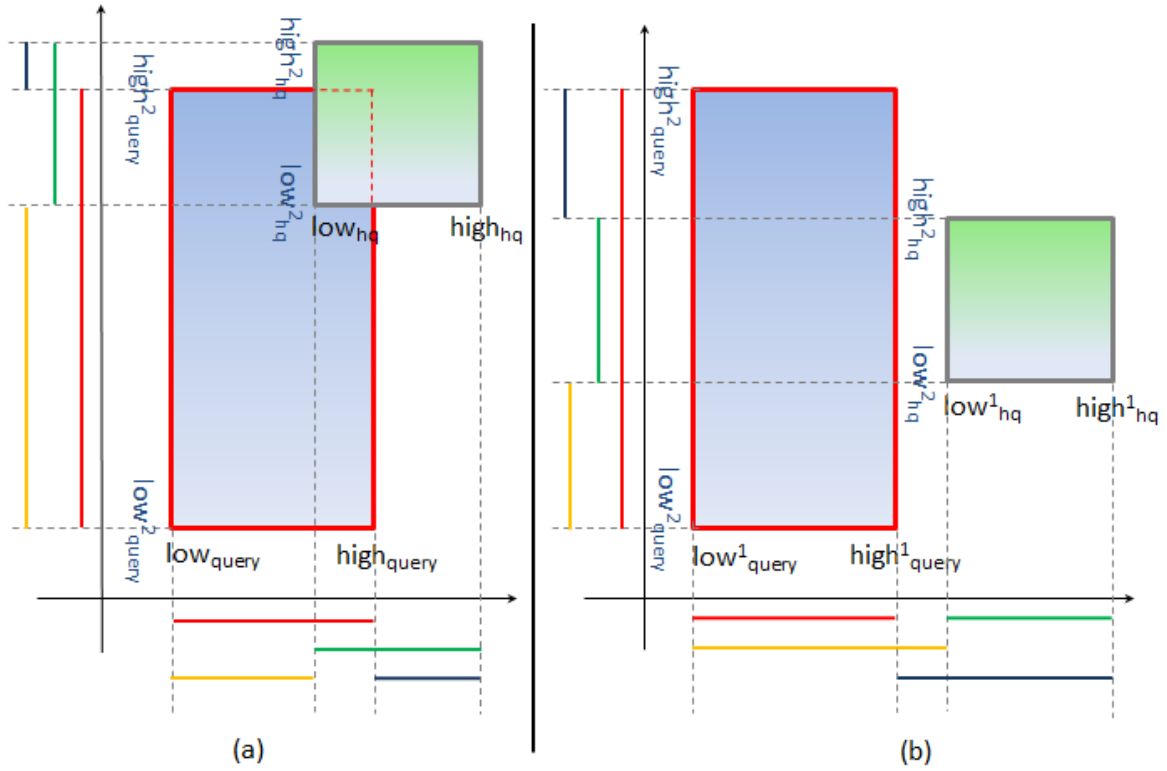


Figura 6.8 – Esempio di intersezione (a) e non (b) dell'iper-quadrante con la range query nel caso bidimensionale

Esempio 6.2.8. Nell'esempio che segue utilizzeremo gli iper-quadranti calcolati nell'esempio 6.2.6 nella sezione 6.2.2, relativi dunque all'intervallo di chiavi derivate [13;55] e cercheremo quali di questi intersecano la seguente range query:

$$1 \leqslant x_1 \leqslant 2$$

$$0 \leqslant x_2 \leqslant 4$$

L'intersezione tra la range query (rettangolo di colore rosso) e i quadranti, che compongono l'intervallo di chiavi derivate [13;55] (colore giallo), è mostrata in figura 6.9.

Per alcuni dei dieci quadranti calcolati dall'algoritmo presentato nella sezione 6.2.1, sono stati ricavati gli intervalli delle coordinate geometriche, che li definiscono, tramite l'algoritmo della sezione 6.2.2 e dunque per ognuno di essi si esegue, per ogni dimensione, il test di intersezione qui sotto nuovamente riportato:

$$|\text{low}^i_{\text{hq}} - \text{low}^i_{\text{query}}| + |\text{high}^i_{\text{hq}} - \text{high}^i_{\text{query}}| \leqslant |\text{low}^i_{\text{hq}} - \text{high}^i_{\text{hq}}| + |\text{low}^i_{\text{query}} - \text{high}^i_{\text{query}}|$$

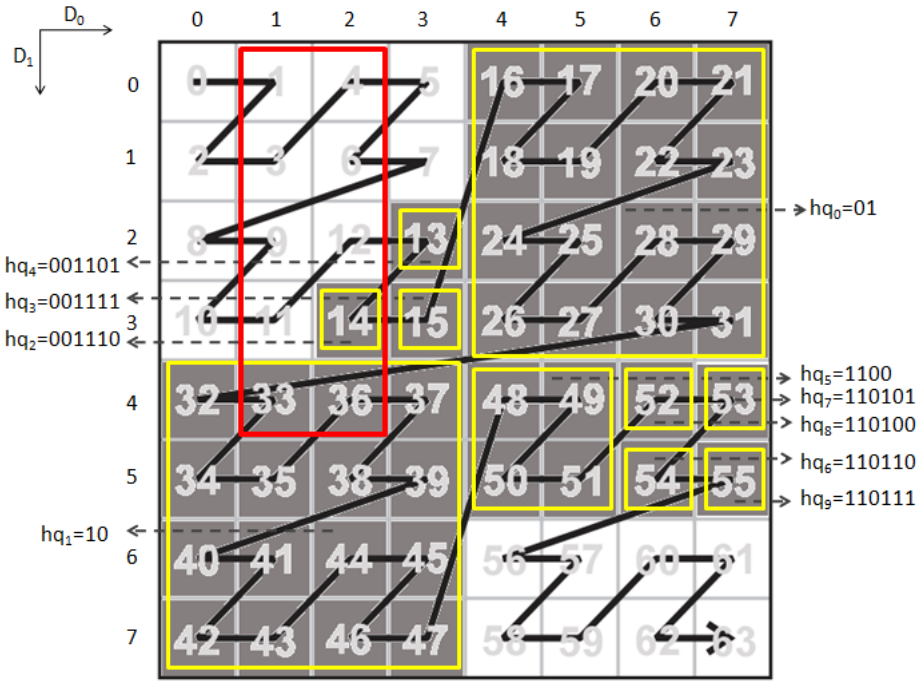


Figura 6.9 – Intersezione tra la range query e i quadranti di output dell'intervallo di chiavi derivate [13;55]

Ad esempio il quadrante $\text{hq}_0 = 01$ è definito dagli intervalli di coordinate

$$4 \leqslant x_0 \leqslant 7$$

$$0 \leqslant x_1 \leqslant 3$$

e dal calcolo della formula di intersezione relativa alla prima dimensione D_0 si ottiene il seguente risultato:

$$|4 - 1| + |7 - 2| \leqslant |4 - 7| + |1 - 2| \Rightarrow 8 \leqslant 4$$

Poichè la disequazione risultante è falsa, è possibile affermare che la range query in questione non interseca il quadrante hq_0 e non è dunque necessario verificare l'intersezione relativa alla seconda dimensione D_1.

Il quadrante $\text{hq}_1 = 10$ è definito dagli intervalli di coordinate

$$0 \leqslant x_0 \leqslant 3$$

$$4 \leqslant x_1 \leqslant 7$$

e dal calcolo della formula di intersezione, relativa alla prima dimensione D_0 si ottiene il seguente risultato:

$$|\,0-1| + |3-2| \leqslant |\,0-3| + |1-2| \Rightarrow 2 \leqslant 4$$

Poichè la disequazione risultante è vera, è necessario controllare se la stessa disequazione è vera anche per la seconda dimensione D_1:

$$|\,4-0| + |7-4| \leqslant |\,4-7| + |0-4| \Rightarrow 7 \leqslant 7$$

La disequazione relativa alla seconda dimensione risulta anch'essa verificata e quindi è possibile affermare che la range query in questione interseca il quadrante hq_1.

Il quadrante $\mathrm{hq}_3 = 001111$ è definito dagli intervalli di coordinate

$$3 \leqslant x_0 \leqslant 3$$

$$3 \leqslant x_1 \leqslant 3$$

e dal calcolo della formula di intersezione, relativa alla prima dimensione D_0 si ottiene il seguente risultato:

$$|\,3-1| + |3-2| \leqslant |\,3-3| + |1-2| \Rightarrow 3 \leqslant 1$$

Poichè la disequazione risultante è falsa, è possibile affermare che la range query in questione non interseca il quadrante hq_3 e non è dunque necessario eseguire il calcolo della formula di intersezione relativa alla seconda dimensione D_1.

6.2.4 Una ottimizzazione

Al fine di diminuire il numero dei test di intersezione degli iper-quadranti con la range query ed evitare il calcolo inutile di alcuni iper-quadranti, è possibile sfruttare una proprietà della *Z-curve* riguardo la distribuzione delle chiavi derivate negli iper-quadranti: dato l'iper-rettangolo che definisce la range query è possibile stabilire i valori della prima chiave derivata α_{query} e dell'ultima chiave derivata β_{query} appartenenti alla regione di *Z-curve* coperta dall'iper-rettangolo. Poichè la chiave derivata α_{query} è associata ai valori minimi ($\text{low}^i_{\text{query}}$) per ciascuno dei vincoli della range query e la chiave derivata β_{query} ai valori massimi ($\text{high}^i_{\text{query}}$) di tali vincoli, è possibile sfruttare tale proprietà e diminuire il numero di test di intersezione confrontando l'intervallo delle chiavi derivate della range query $[\alpha_{\text{query}} ; \beta_{\text{query}}]$ con l'intervallo di chiavi derivate $[\alpha_{\text{digest}} ; \beta_{\text{digest}}]$ ottenuto da una delle strategie di aggregazione di *HASP* presentate nel capitolo 5.

Se vale una elle seguenti condizioni

$$\alpha_{\text{query}} > \beta_{\text{digest}} \qquad (1)$$

$$\beta_{\text{query}} < \alpha_{\text{digest}} \qquad (2)$$

significa che ci troviamo in una di queste due situazioni:

Figura 6.10 – Possibili situazioni di non intersezione tra la range query e una specifica *z-region*

e dunque la range query non interseca nessuno dei quadranti (o iper-quadranti) che descrivono la Z-region compresa tra le chiavi derivate α_{digest} e β_{digest}. Si rivela pertanto non necessaria l'esecuzione dell'algoritmo presentato nella sezione 6.2.1. Si noti come questa proprietà valga per la curva Z-order, ma non ad esempio per la curva di Hilbert.

6.2.5 Complessità dell'algoritmo

L'algoritmo illustrato nelle sezioni 6.2.1, 6.2.2 e 6.2.3 è composto principalmente di due fasi. Nella prima si calcola l'*envelope* di una Z-region e nella seconda la si interseca con l'iper-rettangolo che rappresenta la query.

La prima fase di compone a sua volta di tre passi. Nel primo si individua il quadrante che contiene gli estremi α e β della Z-region, nel secondo si individuano gli iper-quadranti del primo ordine compresi tra quello che contiene α e quello che contiene β e nella terza fase si esaminano ricorsivamente sia il quadrante che contiene α che quello che contiene β. La complessità dei primi due passi è inferiore a quella del terzo e quindi analizziamo solo quest'ultimo.

Il terzo passo visita ricorsivamente l'albero logico dei quadranti individuati dalla suddivisione ricorsiva dello spazio delle chiavi. L'altezza massima di questo albero è k, pari all'ordine della curva, dove $k = (\log_2|D|)$ e D esprime il dominio dei valori degli attributi. Ad ogni passo ricorsivo possono essere presi in considerazione al massimo $2^n - 1$ sotto-quadranti. Ogni sotto-quadrante deve essere intersecato con l'iper-rettangolo che rappresenta la query. In questo passo occorre analizzare le n coordinate dell'iper-rettangolo in cui ciascuna coordinata è rappresentata su $\log_2|D|$ bits, per cui la complessità di questo passo è $O(n{\cdot}\log_2|D|)$. La complessità dell'algoritmo risulta quindi $O(n{\cdot}2^n \cdot \log_2^2|D|)$. Poichè il numero di attributi per la classe di applicazioni da noi considerata è limitato ad una decina, tale complessità risulta accettabile. Nel caso di spazi di attributi più ampi tuttavia, data la complessità esponenziale dell'algoritmo, l'algoritmo può riverlarsi inefficiente.

6.3 La curva di Hilbert

Il calcolo degli iper-quadranti della curva di *Hilbert* che ricoprono una sezione di curva ed intersecano una range query multidimensionale prevede l'esecuzione di complicati test di intersezione tra la curva di Hilbert e la range query. Non è infatti possibile utilizzare una proprietà fondamentale della Z-order curve, cioè il fatto che la curva visita i quadranti mantenendone l'ordinamento. Questa proprietà implica che la chiave derivata coincida con il *navigation code* di un quadrante e ciò semplifica notevolmente gli algoritmi di risoluzione delle range query. Infatti, dato il rettangolo che definisce la query, la chiave derivata massima corrisponde sempre al vertice in alto a sinistra del rettangolo, la minima a quello in basso a destra.

Per la curva di Hilbert non è possibile fare assunzioni su quali siano le chiavi derivate minime e massime contenute in una range query. Esse possono coincidere con un vertice dell'iper-rettangolo che delimita la query o con un altro punto qualsiasi. Ad esempio, nella query mostrata in figura 6.11(a), la chiave derivata di valore minimo

coincide con il vertice in basso a destra dell'iper-rettangolo definito dalla query, mentre quella massima con un punto sul lato di questo iper-rettangolo. E' possibile notare come non vi sia alcuna intersezione tra i valori delle chiavi derivate che soddisfano i vincoli di una range query sulla curva di Hilbert. I segmenti della curva di Hilbert che soddisfano la range query, in special modo per l'esempio in figura 6.11(a), non esplicitano alcuna proprietà sfruttabile per i valori delle chiavi derivate che li compongono, in quanto, proprio per la natura stessa della curva di Hilbert che entra ed esce ripetutamente dall'iper-rettangolo che delimita la range query, l'individuazione dei clusters di chiavi derivate è resa molto complicata. Lo sviluppo degli algoritmi per il supporto di range query sulla curva space-filling di Hilbert sarà oggetto degli sviluppi futuri di questa tesi.

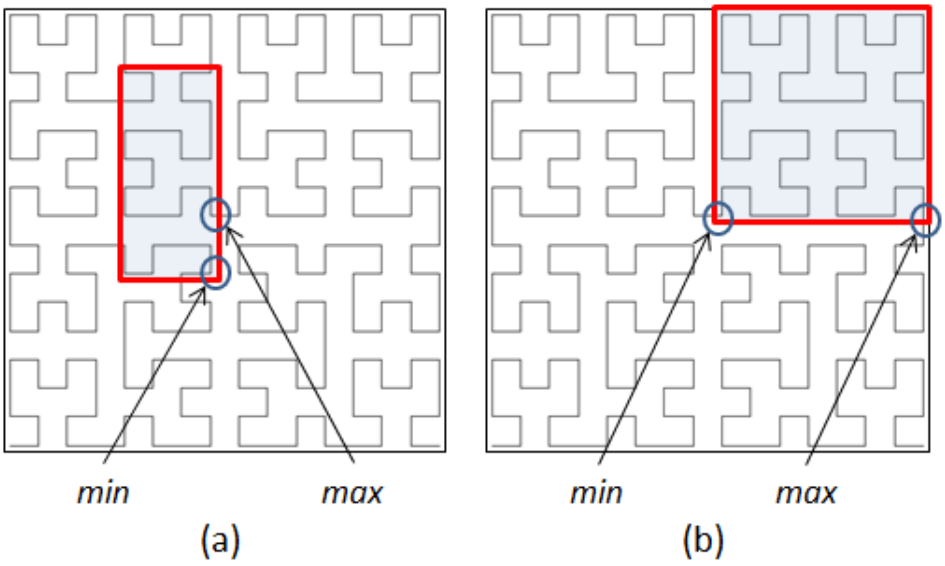


Figura 6.11 – Esempi di intersezione di query con la curva di Hilbert del terzo ordine

Capitolo 7

HASP: l'implementazione

In questo capitolo verrà presentata l'implementazione di *HASP*. *HASP* è stato implementato mediante l'uso del framework *Overlay Weaver,* descritto nella sezione successiva. Nelle sezioni successive invece verrà descritto il processo implementativo seguito per la definizione delle nuove strategie di digest presenti in *HASP* e sarà riportata l'implementazione degli algoritmi presentati nei capitoli 5 e 6.

7.1 Framework Overlay Weaver

Lo sviluppo di *HASP* si basa sull'utilizzo e sulla estensione del framework *Overlay Weaver* [41]. Overlay Weaver è un framework *Open Source* realizzato come lavoro di ricerca e reperibile gratuitamente in rete. Le caratteristiche principali del framework consistono nella elevata *modularità* e *strutturazione a livelli*. Questo consente di realizzare applicazioni di rete in grado di appoggiarsi a moduli esistenti e di ereditarne un'elevata semplicità di configurazione. Il forte disaccoppiamento del framework consente infatti di effettuare agevolmente delle variazioni del comportamento dell'applicazione. OW supporta diversi overlay strutturati tra cui Chord [8], Pastry [10], Tapestry [11], Kademlia [14] e Koorde. Inoltre fornisce, come applicazioni ad alto livello, una semplice shell DHT interattiva e una shell Multicast. Il framework mette a disposizione anche una serie di tools per lo sviluppo e per il debugging come l'emulatore di nodi, il generatore di scenari ed un visualizzatore grafico della topologia della rete.

7.1.1 Architettura del framework

L'architettura di *OverlayWeaver*, come già accennato, risulta fortemente strutturata a livelli ed in particolare possiamo individuare i seguenti livelli:

- Applicazioni
- Servizi ad alto livello
- Servizi di routing
- Servizi di memorizzazione

La figura 7.1 riassume graficamente l'architettura e consente di poter osservare le relazioni di dipendenza tra i vari livelli. Il livello delle applicazioni utilizza esclusivamente i servizi disponibili ad alto livello, analogamente i servizi ad alto livello si poggiano sui servizi di routing e di memorizzazione. Per quanto riguarda il livello di routing, *Overlay Weaver* effettua un'ulteriore suddivisione basata sul concetto di *Key-based routing*, *KBR*, introdotto da [42] che generalizza le funzionalità delle principali DHT. In generale, in una rete strutturata, ogni nodo memorizza un sotto-insieme di collegamenti ad altri nodi e la scelta di tali nodi identifica la topologia della rete. Per effettuare la ricerca di una chiave, il modello *KBR* consente di indirizzare la ricerca verso dei nodi via via sempre più vicini al nodo *target* in accordo ad una *metrica*. *Overlay Weaver* basandosi su tale modello suddivide ulteriormente il livello di routing in:

- ***Routing Driver***, espone le principali operazioni basate sul modello *KBR*.
- ***Routing Algorithm***, contiene le diverse strategie di routing.
- ***Messaging Service***, contiene diverse strategie di comunicazione di rete.

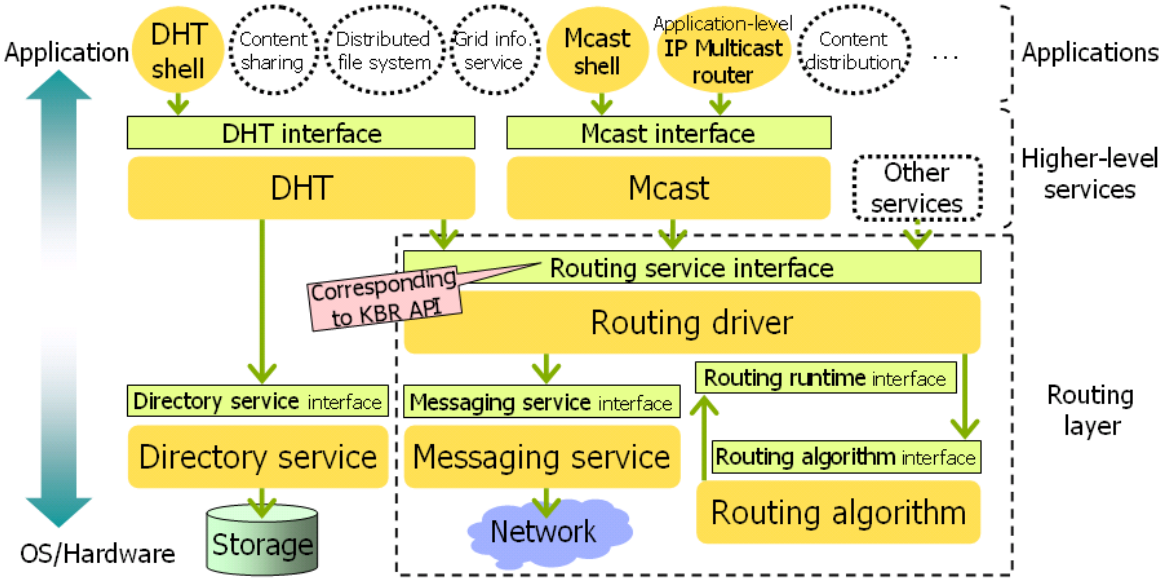


Figura 7.1 – Architettura di Overlay Weaver

Come implementazione del livello di *Routing Driver*, *Overlay Weaver* fornisce due versioni, una iterativa ed una ricorsiva. La figura 7.2 illustra le diverse comunicazioni effettuate dai diversi tipi di routing.

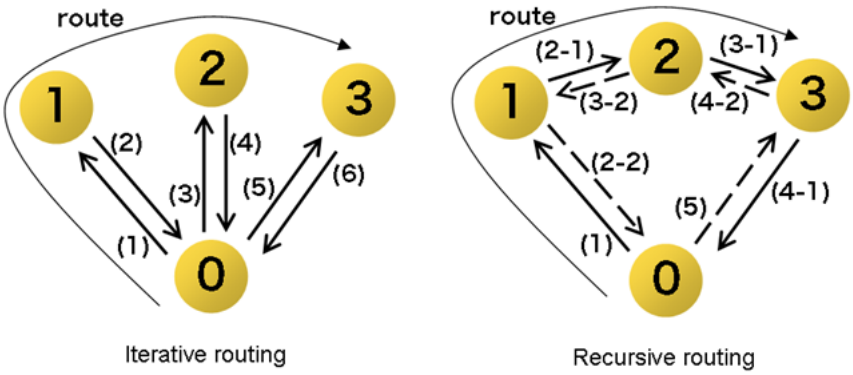


Figura 7.2 – Tipologie di routing in Overlay Weaver

Le linee identificano i messaggi scambiati tra i vari nodi mentre i numeri annotati l'ordine di scambio. La notazione tratteggiata sta ad indicare i messaggi che possono non essere necessariamente scambiati durante il routing, ma che servono in caso di

utilizzo di protocolli non affidabili, quale ad esempio UDP. Per il livello di *Routing Algorithm*, *OW* implementa le strategie di routing Chord, Pastry, Tapestry, Kademlia e Koorde. Nel livello di *Messaging Service* sono implementate le classi per la comunicazione di rete attraverso i protocolli TCP, UDP e intra-thread utilizzato nel caso di emulazione. Nel livello di memorizzazione *OW* offre la memorizzazione attraverso un database relazionale (Berkeley DB) o in alternativa la memorizzazione in memoria principale attraverso le hash table della *Java standard class library*.

7.1.2 Tools di sviluppo

Overlay Weaver mette a disposizione una serie di strumenti di sviluppo aggiuntivi come l'emulatore di rete, il generatore di scenari e un visualizzatore grafico di rete. Lo strumento di maggior utilizzo risulta essere l'*emulatore*. Attraverso l'emulatore è possibile testare le applicazioni in maniera immediata in un ambiente controllato. L'emulatore prevede due modalità di esecuzione: la modalita locale in cui i nodi emulati sono localizzati nella sola macchina fisica locale e la modalita distribuita in cui varie istanze di emulatori sono attive in macchine fisiche distribuite e la comunicazione tra i rispettivi nodi emulati avviene attraverso la rete. Per quanto riguarda la modalità di interazione con lo strumento possiamo distinguere la modalità interattiva e la modalità *batch*. Nel primo caso i comandi sono impartiti all'emulatore attraverso una console testuale, mentre nel secondo caso viene fornito un file di testo o scenario contenente una serie di comandi in grado di essere interpretati ed eseguiti dall'emulatore.

La figura 7.3 illustra la struttura tipica di uno scenario. In particolare è possibile notare le relative direttive in grado di eseguire una serie ripetuta di comandi e una sequenza di comandi in grado di temporizzare l'esecuzione. Infine la creazione di uno scenario può essere assistita da un ulteriore tool presente nel framework denominato *generatore di scenari*. L'ultimo strumento presente nel framework è il visualizzatore grafico di rete visibile in figura 7.4. Il visualizzatore consente di rappresentare, in varie modalità grafiche, la locazione dei nodi nella rete e di visualizzare in tempo reale le comunicazioni tra i nodi. In particolare è possibile distinguere graficamente le varie tipologie di messaggi scambiati.

```
timeoffset 2000

#invoca il primo nodo
class.ow.tool.dhtshell.main
arg -p 10000
schedule 0 invoke

#invoca 3 nodi
arg
schedule 1000,1000,3 invoke

timeoffset 7000

#i 3 nodi entrano nell'overlay
schedule 0 control 1 init emu0
schedule 1000 control 2 init emu0
schedule 2000 control 3 init emu0

#inserisce la chiave di un nodo: put
schedule 4000 control 1 setdynamic key1 value1

#operazione di ricerca di una chiave: get
schedule 5000 control 2 getdynamic key1
```

Figura 7.3 – Esempio di scenario interpretabile dall'emulatore

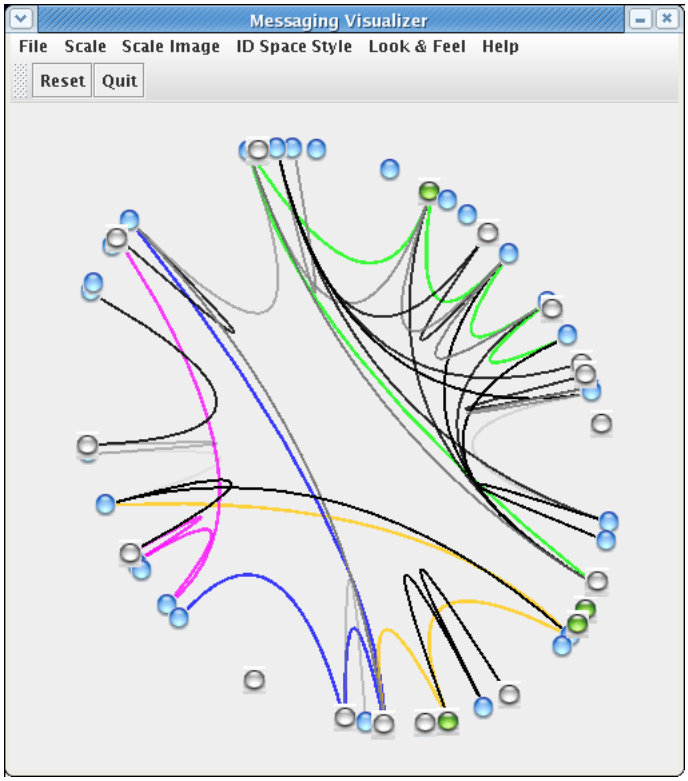


Figura 7.4 – Esempio di visualizzazione grafica della rete

7.2 Definizione di uno scenario

L'emulazione di un numero N di nodi in locale necessita la definizione di un file di scenario contenente i vari comandi che l'interprete principale del sistema *HASP* deve eseguire. L'interprete principale del sistema è definito nella classe *ow.tool.xconeshell.main.java* e contiene la definizione dei vari comandi che possono essere invocati. Un file di scenario è composto da quattro sezioni principali:

1. definizione ed inizializzazione degli N nodi del sistema
2. definizione dell'associazione delle risorse ai nodi del sistema
3. fase di join dei nodi sulla rete
4. definizione delle range query da eseguire sulla rete locale creata

In figura 7.5 è mostrata la prima parte di un possibile scenario. Si può notare come venga eseguito il comando *invoke* sul nodo 0, il nodo di *bootstrap* e poi, dopo 2 secondi da questo, su altri 99 nodi a distanza di 250ms l'uno dall'altro. Il comando invoke viene interpretato dall'interprete di *HASP* ed inizializza tutte le strutture dati di un nodo.

```
###################################
# SET-UP VIRTUAL NODES
###################################
# invokes the 1st XCone shell
class ow.tool.xconeshell.Main
schedule 0 invoke

# invokes others nodes
schedule 2000,250,99 invoke

# wait structures inizialization
timeoffset +5000
```

Figura 7.5 – *Scenario Sezione I*: definizione ed inizializzazione dei nodi del sistema

In figura 7.6 è mostrata la seconda sezione di un possibile scenario. In questa parte viene effettuata l'assegnazione delle risorse ai nodi mediante il comando *setdynamic*. All'interno del comando *setdynamic* è definita la curva space-filling che, a partire dagli attributi che compongono la risorsa del nodo, permette la *generazione della chiave derivata*.

```
###################################
# SET-UP KEYS
###################################
# set dynamic attributes

schedule 0 control 0 setdynamic freeSwap 11 - cpuUse 2
schedule 1 control 1 setdynamic freeSwap 4 - cpuUse 7
schedule 2 control 2 setdynamic freeSwap 8 - cpuUse 0
schedule 3 control 3 setdynamic freeSwap 9 - cpuUse 4
schedule 4 control 4 setdynamic freeSwap 3 - cpuUse 4
schedule 5 control 5 setdynamic freeSwap 0 - cpuUse 14
schedule 6 control 6 setdynamic freeSwap 0 - cpuUse 4
schedule 7 control 7 setdynamic freeSwap 12 - cpuUse 14
schedule 8 control 8 setdynamic freeSwap 7 - cpuUse 2
schedule 9 control 9 setdynamic freeSwap 0 - cpuUse 11

# wait structures inizialization
timeoffset +5000
```

Figura 7.6 – *Scenario Sezione II*: specifica ed assegnazione delle risorse ai nodi

In figura 7.7 è mostrata la terza sezione di un possibile scenario nella quale viene effettuata la *join* dei nodi sulla rete *HASP* mediante il comando *init*. Il comando *init* esegue l'inserzione del nodo sulla DHT e nell'albero logico *HASP*.

```
###################################
# SET-UP XCONE OVERLAY
###################################
# start nodes join on DHT overlay

schedule    0 control 0 init emu0
schedule  500 control 1 init emu0
schedule 1000 control 2 init emu0
schedule 1250 control 3 init emu2
schedule 1500 control 4 init emu3
schedule 1750 control 5 init emu2
schedule 2000 control 6 init emu2
schedule 2250 control 7 init emu4
schedule 2500 control 8 init emu7
schedule 2750 control 9 init emu7

timeoffset +30000
```

Figura 7.7 – *Scenario Sezione III*: *join* dei nodi sulla rete

In figura 7.8 è mostrata la quarta ed ultima sezione di un possibile scenario. In questa parte vengono definite le range query da eseguire sulla rete *HASP* mediante il comando *getdynamic*. Il comando *getdynamic* assegna ad un nodo l'esecuzione di un range query.

```
##################################
# TESTS
##################################

schedule 0 control 0 getdynamic freeSwap 5:12 cpuUse 8:12 5
schedule 0 control 1 getdynamic freeSwap 1:6 cpuUse 4:10 2
schedule 0 control 2 getdynamic freeSwap 8:12 cpuUse 10:12 4
schedule 0 control 3 getdynamic freeSwap 2:7 cpuUse 3:6 7
schedule 0 control 4 getdynamic freeSwap 7:11 cpuUse 2:5 3
```

Figura 7.8 – *Scenario Sezione IV:* definizione delle range query

7.3 Implementazione delle curve space-filling

L'elevata modularità del sistema XCone ha permesso l'introduzione delle curve space-filling e delle relative chiavi derivate in maniera particolarmente semplice. Le modifiche che sono state effettuate riguardano il modulo *KeyManager* in *ow.src.xcone.resource*. Tale modulo riceve dal file di scenario gli n attributi che definiscono le risorse di ciascun nodo, costruisce l'oggetto *ZCurve*, o *HilbertCurve,* che rappresenta la curva space-filling e a partire dal valore degli attributi calcola la chiave derivata da inserire nel sistema. All'interno della classe *ZCurve.java* in *ow.src.xcone.impl.sfc*, sono contenuti gli algoritmi per la generazione della chiave derivata e per la risoluzione di una range query per la curva Z-order descritti rispettivamente nelle sezioni 4.2 e 6.2. L'algoritmo per la generazione della chiave derivata di Hilbert descritto nella sezione 4.1.2 è stato implementato ed integrato in *HASP* nel modulo *KeyManager,* ma affinchè sia possibile utilizzare tale curva in *HASP* è necessaria l'implementazione dell'algoritmo di risoluzione di una range query sulla curva di Hilbert, il quale sarà oggetto degli sviluppi futuri della presente tesi.

7.4 Implementazione del q-digest dinamico

L'elevata modularità di XCone ha permesso la definizione in *HASP* delle nuove strutture di digest presentate nel capitolo 5. Per ciascuna delle nuove tecniche di digest realizzate per il supporto a range query multidimensionali è stata necessaria l'implementazione dell'interfaccia *DigestInfo* e della classe astratta *DigestStrategy* presenti in XCone. L'interfaccia *DigestInfo* rappresenta l'informazione di digest utilizzata nelle operazioni in XCone e richiede che ciascuna tecnica di digest implementi i metodi:

- *public String getSerialized()*
- *public boolean equals(DigestInfo v1)*

Il metodo *getSerialized* è necessario per la serializzazione dell'oggetto che contiene l'informazione di digest in modo che possa essere inviato ai vari nodi della rete.

La classe astratta *DigestStrategy* esprime le operazioni eseguibili su uno o più oggetti che contengono l'informazione di digest e che devono essere necessariamente implementate:

- *public abstract DigestInfo aggregate(DigestInfo v1, int level1, DigestInfo v2, int level2)*
- *public abstract int estimate(XConeQuery xQuery, DigestInfo value, int level)*
- *public abstract DigestInfo getDigestInfo(BigInteger value, int level)*
- *public abstract DigestInfo deserializeDigestInfo(String value)*

Il metodo *aggregate* è l'operazione necessaria affinchè due informazioni di digest provenienti da nodi differenti dell'albero XCone, e ricevuti come parametri, possano essere unite in un'unico oggetto che contiene le informazioni di digest di entrambi.

Il metodo *estimate* riceve come parametri una range query e un'informazione di digest e restituisce il numero di risorse presenti nell'informazione di digest che soddisfano tale range query.

Il metodo *getDigestInfo* riceve come parametro la chiave del nodo e da questa crea l'oggetto che contiene l'informazione di digest.

Il metodo *deserializeDigestInfo* è il duale del metodo *getSerialized()* descritto in precedenza per l'interfaccia *DigestInfo*. Questo metodo riceve, come parametro in ingresso, una stringa risultato di un'operazione di serializzazione dell'informazione di digest di altro nodo della rete e a partire da tale stringa ricostruisce l'oggetto che contiene l'informazione di digest di quel nodo.

Una delle strategie di digest presenti in *HASP* utilizza una struttura dati dinamica per memorizzare l'albero di *q-digest*. Tale struttura dati è descritta in precedenza nel capitolo 5, sezione 5.2.4. Poichè ciascuna strategia di digest deve implementare le interfacce *DigestInfo* e *DigestStrategy* descritte nel paragrafo precedente, anche la tecnica

di digest *q-digest,* che utilizza una struttura dinamica per mantenere le informazioni, deve farlo.

Le classi che definiscono questa strategia di digest in *HASP* sono:

- *QDigestTree* che contiene la rappresentazione dell'albero di *q-digest*. L'albero di *q-digest* è rappresentato tramite una *HashMap* che contiene le informazioni sui vari livelli dell'albero ed indicizzata tramite il livello.
- *QDigestLevel* che contiene una lista di elementi per ogni livello dell'albero di *q-digest*
- *QDigestElement* che contiene la rappresentazione del singolo nodo di *q-digest*.

Ciascun nodo dell'albero esprime un intervallo di digest [*low*; *up*] ed è caratterizzato dai seguenti campi:

- valore *low* dell'intervallo
- valore *up* dell'intervallo
- *count*, ovvero il numero di chiavi derivate contenute nell'intervallo

La classe *QDigestStrategy* implementa la classe astratta *DigestStrategy*. Il metodo *aggregate* unisce due alberi di *q-digest* provenienti da nodi differenti della rete ed in questa nuova tecnica di digest contiene l'algoritmo di fusione di due alberi di *q-digest* e di compressione di un albero di *q-digest* presentati nella sezione 5.2.4. Il metodo *estimate* deve restituire il numero di chiavi derivate che soddisfano una determinata range query e contiene dunque al suo interno l'algoritmo del processo di risoluzione delle range query multidimensionali descritto nella sezione 6. Tale algoritmo viene eseguito per ogni nodo dell'albero di *q-digest*. Una volta esaminati tutti i nodi della struttura dati dinamica che rappresenta l'albero di *q-digest,* si ottiene il numero di chiavi derivate che soddisfano la range query passata come parametro. La classe *QDigestTree* implementa l'interfaccia *DigestInfo*. La serializzazione dell'albero di *q-digest* avviene tramite la concatenazione delle stringhe risultato delle serializzazioni dei nodi presenti nell'albero. Ciascun elemento, *QDigestElement*, dell'albero di *q-digest* viene serializzato nel seguente formato:

livello sep *lowValue* sep *highValue* sep *count*

in cui *sep* indica un carattere separatore dei vari campi dell'elemento. Il processo di deserializzazione, a partire dalla stringa ricevuta, ricava i vari elementi effettuando un'operazione di split della stringa tramite il carattere separatore e li aggiunge ad un nuovo albero di *q-digest*, ricreando così sul nodo destinatario la struttura dati presente sul nodo mittente. Il metodo *getDigestInfo* in questa strategia di digest costruisce un albero di *q-digest* con un solo nodo corrispondente alla chiave derivata che possiede. In figura 7.9 è mostrato il diagramma delle classi definite per la strategia di aggregazione delle chiavi *q-digest*.

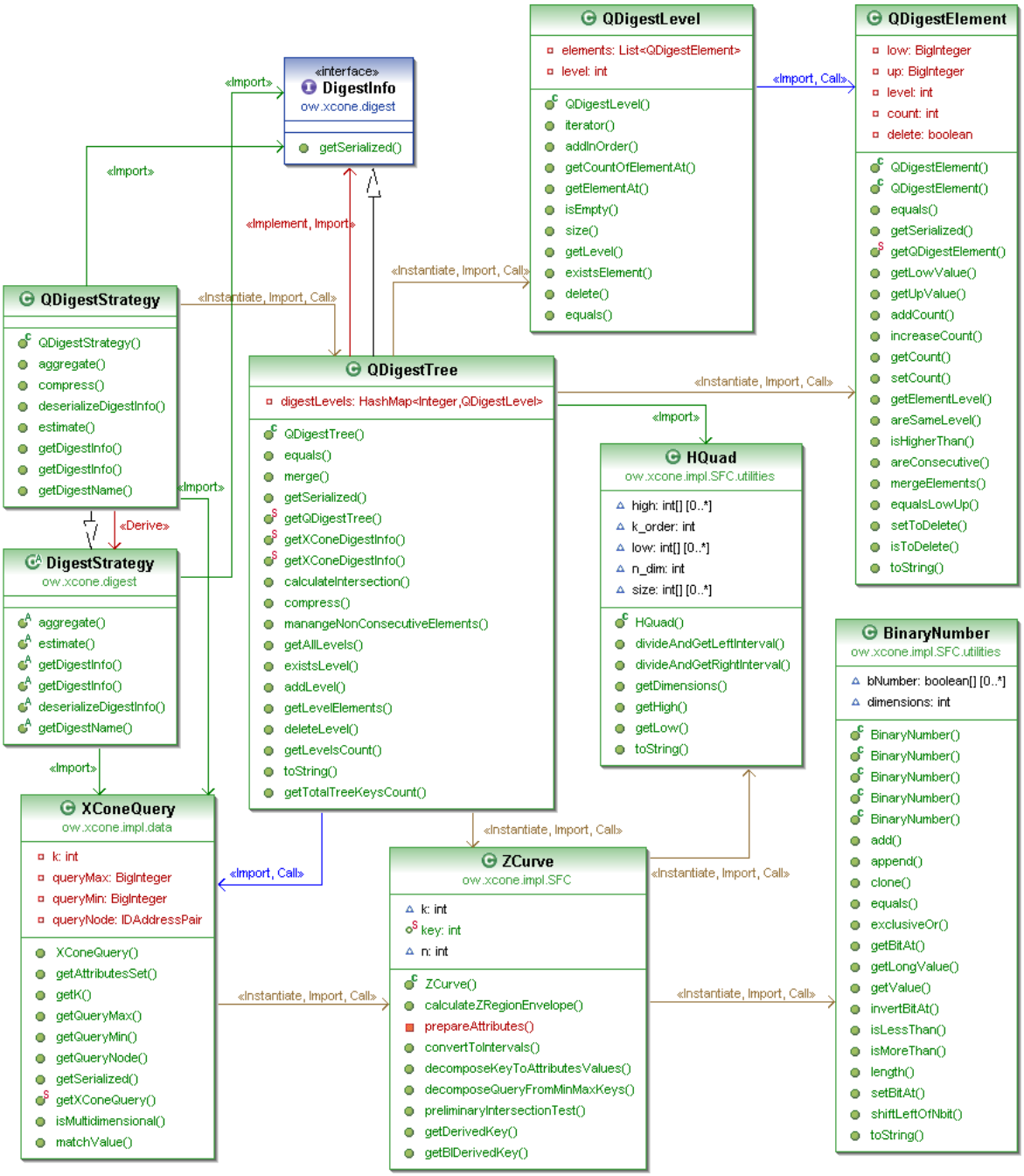


Figura 7.9 – Diagramma delle classi del *q-digest* con struttura dati dinamica in *HASP*

7.5 Pseudocodice degli algoritmi

In questa sezione è mostrato lo pseudocodice degli algoritmi descritti nei capitoli 5 e 6, rispettivamente per le operazioni sull'albero di *q-digest* rappresentato con una struttura dati dinamica e per il processo di risoluzione di range query multidimensionali.

7.5.1 Algoritmi per la struttura dati dinamica *q-digest*

Le due operazioni principali definite sull'albero dinamico di *q-digest* e descritte nella sezione 5.2.4 sono le operazioni di compressione dell'albero e di fusione di due alberi di *q-digest*.

Codice 7.1 – Algoritmo di compressione dell'albero dinamico di q-digest

```
1  compress(level,n,k){
2     i=0;
3     /* si ricava la lista di tutti gli elementi appartenenti al livello level */
4     levelElementsList = getLevelElementsList(level);
5     while (i<levelElementList.size()) do {
6        node1 = levelElementList.getElementAtIdx(i);
7        node2 = levelElementList.getElementAtIdx(i+1);
8        /* controllo se due nodi sono consecutivi ovvero se low2 = up1+1 */
9        if (areConsecutiveNodes(node1,node2)) then {
10          mergeNode = merge(node1,node2);
11          /* controllo se due nodi consecutivi, danno origine ad un nodo di livello inferiore valido */
12          validMerge = checkMerge(mergeNode,level-1);
13          if (validMerge) then {
14             /* l'unione dei nodi e' valida, controllo se il padre esista gia' nell'albero */
15             if (nodeExistsAtLevel(mergeNode,level-1)) then {
16                fatherNode =  getElementAtLevel(mergeNode, level-1);
17                /* controllo se la somma dei count del nodo padre esistente con quello due nodi uniti verifica o meno */
18                /* la proprieta' di QDigest */
19                qDigestCheck = checkQDigestProperty(mergeNode.getCount() +  fatherNode.getCount(),n,k);
20                if (qDigestCheck = notVerified) then {
21                   /* se non la verifica, i due nodi sono rimossi dall'albero e il loro count aggiunto a quello del padre */
22                   removeNodeFromLevel(node1,level);
23                   removeNodeFromLevel(node2,level);
24                   fatherNode.updateCount(fatherNode.getCount() + mergeNode.getCount());
25                }
26             }
27             else{
28                /* l'unione dei due nodi e' valida, ma il padre non esiste nell'albero */
29                /* controllo proprieta' q-digest con il solo count dell'unione dei due nodi */
30                qDigestCheck = checkQDigestProperty(mergeNode.getCount() + fatherNode.getCount(),n,k);
31                if (qDigestCheck = notVerified) then {
32                   /* se non la verifica, i due nodi sono rimossi dall'albero e il loro padre aggiunto al livello */
33                   /* inferiore con un count pari alla somma dei loro count */
34                   removeNodeFromLevel(node1,level);
35                   removeNodeFromLevel(node2,level);
36                   insertNodeAtLevel(mergeNode,level-1);
37                }
38             }
39             i = i+2;
40          }
41          else{
42             /* l'unione dei due nodi consecutivi non ha prodotto un merge valido, si considera solo il primo elemento */
43             /* si calcola il nodo padre del primo elemento e si controlla se e' gia' presente nell'albero */
44             fatherNode = node1.calculateFatherNode();
45             if (nodeExistsAtLevel(fatherNode,level-1)) then {
46                /* controllo proprieta' q-digest con la somma dei count del nodo 1 e del nodo padre */
```

```
47                qDigestCheck = checkQDigestProperty(node1.getCount() + fatherNode.getCount(),n,k);
48                if (qDigestCheck = notVerified) then {
49                   /* se non la verifica, il nodo 1 e' rimosso dall'albero e il count del padre aggiornato */
50                   removeNodeFromLevel(node1,level);
51                   fatherNode.updateCount(fatherNode.getCount() + node1.getCount());
52                }
53             }
54             else{
55                /* il padre del nodo 1 non esiste nell'albero*/
56                /* controllo proprieta' q-digest con il solo count del nodo 1 */
57                qDigestCheck = checkQDigestProperty(node1.getCount(),n,k);
58                if (qDigestCheck = notVerified) then {
59                   /* se non la verifica, il nodo 1 e' rimosso dall'albero e il suo nodo padre aggiunto al livello */
60                   /* inferiore con un count pari a quello del nodo 1 */
61                   fatherNode.setCount(node1.getCount());
62                   removeNodeFromLevel(node1,level);
63                   insertNodeAtLevel(fatherNode,level-1);
64             }
65             i++;
66          }
67       }
68       else{
69          /* i due nodi non sono consecutivi, si considera solo il nodo 1 */
70          /* si calcola il nodo padre del primo elemento e si controlla se e' gia' presente nell'albero */
71          fatherNode = node1.calculateFatherNode();
72          if (nodeExistsAtLevel(fatherNode,level-1)) then {
73             /* controllo proprieta' q-digest con la somma dei count del nodo 1 e del nodo padre */
74             qDigestCheck = checkQDigestProperty(node1.getCount() + fatherNode.getCount(),n,k);
75             if (qDigestCheck = notVerified) then {
76                /* se non la verifica, il nodo 1 e' rimosso dall'albero e il count del padre aggiornato */
77                removeNodeFromLevel(node1,level);
78                fatherNode.updateCount(fatherNode.getCount() + node1.getCount());
79             }
80          }
81          else{
82             /* il padre non esiste nell'albero di QDigest */
83             /* controllo proprieta' q-digest con il solo count del nodo 1 */
84             qDigestCheck = checkQDigestProperty(node1.getCount(),n,k);
85             if (qDigestCheck = notVerified) then {
86                /* se non la verifica, il nodo 1 e' rimosso dall'albero e il suo nodo padre aggiunto al livello */
87                /* inferiore con un count pari a quello del nodo 1 */
88                fatherNode.setCount(node1.getCount());
89                removeNodeFromLevel(node1,level);
90                insertNodeAtLevel(fatherNode,level-1);
91             }
92             i++;
93          }
94       }
95    }
96 }
```

Codice 7.2 – Algoritmo per il merge di due alberi di q-digest

```
1  merge(QDigestTree1,QDigestTree2){
2     /* per ciascuno dei livelli degli alberi */
3     for (i=totalLevels; i>0; i++){
4        /* se un livello e' presente in entrambi gli alberi, si crea un nuovo livello come merge dei due singoli livelli */
5        if (QDigestTree1.existsLevel(i) && QDigestTree2.existsLevel(i)) then {
6           levelTree1 = QDigestTree1.getLevel(i);
7           levelTree2 = QDigestTree2.getLevel(i);
8           mergelevel = mergeLevels(levelTree1,levelTree2);
9           newTree.addLevel(mergeLevel);
10       }
11       else{
12          /* se un livello e' presente solo nel primo dei due alberi di q-digest, si aggiunge il livello */
13          /* senza modifiche nel nuovo albero */
14          if (QDigestTree1.existsLevel(i)) then {
15             levelTree1 = QDigestTree1.getLevel(i);
16             newTree.addLevel(levelTree1);
17          }
18          else{
19             /* se un livello e' presente solo nel secondo dei due alberi di q-digest, si aggiunge il livello */
20             /* senza modifiche nel nuovo albero */
21             levelTree2 = QDigestTree2.getLevel(i);
22             newTree.addLevel(levelTree2);
23          }
24       }
25    }
26    /* si esegue l'algoritmo di compressione su tutti i livelli del nuovo albero */
27    for (i=totalLevels; i>1; i++){
28       compress(newTree.getLevel(i),n,k);
29    }
30 }
```

7.5.2 Algoritmi per la risoluzione di range query multidimensionali

Il processo di risoluzione di range query multidimensionali descritto nel capitolo 6 si basa su due algoritmi principali:

- l'algoritmo per il calcolo della *Z-region envelope* tra due chiavi derivate α e β, descritto nella sezione 6.2.1
- l'algoritmo per il calcolo delle coordinate geometriche degli iper-quadranti restituiti dall'algoritmo precedente descritto nella sezione 6.2.2.

Codice 7.3 – Algoritmo per il calcolo della ZRegion envelope compresa tra due chiavi derivate alpha e beta

```
1  calculateZRegion(key alpha, key beta){
2        /* si calcolano le codifiche binarie delle chiavi alpha e beta su n*k bit */
3        alphaBin = getBinaryRepresentation(alpha);
4        betaBin = getBinaryRepresentation(beta);
5        currentAlphaPrefix = null;
6        currentBetaPrefix = null;
7        /* si calcola (se esiste) un prefisso comune tra le due chiavi
8        commonPrefix = getKeysCommonPrefix(alphaBin,betaBin);
9        /* se il prefisso comune ha lunghezza maggiore del numero di attributi significa che le */
10       /* due chiavi derivate possiedono almeno un quadrante in comune */
11       if(commonPrefix.length > n) then {
12             /* si calcola il numero di gruppo di n bit in comune tra le due chiavi */
13             groupsCount =  commonPrefix/n;
14             /* si trova il quadrante minimo che contiene entrabe le chiavi derivate */
15             minHQuad = getDigitsFromTo(alphaBin,0,groupsCount*n);
16             /* e si aggiunde ai quadranti in output che formano la Z-Region compresa tra le chiavi */
17             output.add(minHQuad);
18             /* si setta il prefisso corrente sia di alpha che di beta al valore dell'iper-quadrante minimo comune */
19             currentAlphaPrefix = minHQuad;
20             currentBetaPrefix = minHQuad;
21       }else groupsCount = 0;
22       /* si scorrono i bit di alpha e beta a gruppi di n alla volta */
23       for(i=(groupsCount*n); i<alphaBin.length; i=i+n){
24             /* si esaminano i successivi n bit della chiave alpha */
25             examAlpha = getDigitsFromTo(alphaBin,i,i+n);
26             /* e si aggiungono in output tutti gli iper-quadranti maggiori dell'iper-quadrante
27             /* formato dal prefisso corrente e dai bit in esame di alpha */
28             output.add(calculateHQuadsGreaterThan(currentAlphaPrefix + examAlpha));
29             /* si esaminano i successivi n bit della chiave alpha */
30             examBeta = getDigitsFromTo(betaBin,i,i+n);
31             /* e si aggiungono in output tutti gli iper-quadranti minori dell'iper-quadrante
32             /* formato dal prefisso corrente e dai bit in esame di beta */
33             output.add(calculateHQuadsSmallerThan(currentBetaPrefix + examBeta));
34             /* si aggiornano i prefissi correnti di alpha e beta aggiungendovi i bit appena esaminati */
35             currentAlphaPrefix = currentAlphaPrefix + examAlpha;
36             currentBetaPrefix = currentBetaPrefix + examBeta;
37       }
38       /* al termine si aggiungono alla lista degli iper-quadranti compresi tra le chiavi derivate */
39       /* alpha e beta (output), le stesse chiavi derivate alpha e beta */
40       output.add(alphaBin);
41       output.add(betaBin);
42 }
```

Codice 7.4 – Algoritmo per la conversione del navigation code di un iper-quadrante nelle coordinate geometriche che lo definiscono

```
convertHQuadToAttributesIntervals(HQuad hq,n,k){
      /* si inizializzano gli intervalli relativi agli n attributi pari all'intervallo massimo */
      /* n = numero di attributi (o dimensioni)
      /* k = ordine della curva (massimo valore assumibile da ciascun attributo */
      /*     e pari a (2^k)-1 */
      for (i=0; i<n; i++){
            interval[i] =  [0;(2^k)-1]
      }
      /* si scorrono tutti i bit che compongono la codifica binaria dell'iper-quadrante hq */
      for (j=0; j<hq.size(); j++){
            bitValue = getBitAtPosition(hq,j);
            /* si ottiene l'indice dell'attributo relativo a tale bit j-esimo */
            attribute =  i mod (n-1);
            /* se il bit ha valore zero, si divide a meta' l'intervallo e si prende la parte sinistra */
            if (bitValue = 0) then {
                  interval[attribute] = [0;(2^(k-1)-1];
            }
            /* altrimenti si prende la parte destra */
            else{
                  interval[attribute] = [2^(k-1);(2^k)-1];
            }
      }
}
```

Capitolo 8

Risultati sperimentali

Questo capitolo è formato da tre parti principali. Nella sezione 8.1 verrà descritta *PlanetLab*, la piattaforma da cui sono stati raccolti i dati reali utilizzati per i test. Nella sezione 8.2 sarà descritta la metodologia di definizione dei test effettuati ed infine, nella sezione 8.3, verranno riportati i risultati sperimentati ottenuti dal sistema *HASP*.

8.1 PlanetLab

PlanetLab [43] è una piattaforma aperta per lo sviluppo, il *deployment* e l'accesso a servizi su scala mondiale mediante la rete Internet. La rete dei nodi PlanetLab è, ad oggi, composta da 1138 nodi distribuiti in 519 siti in tutto il mondo. Tale infrastruttura è utilizzata per l'esecuzione parallela di job che vengono schedulati quindi su nodi localizzati su scala mondiale. Gli *host* che fanno parte della rete PlanetLab inviano periodicamente il loro stato ai server, in modo che essi possano controllare lo stato di congestione dei nodi della rete nel tempo e di conseguenza distribuire efficacemente il carico di lavoro sulle varie macchine. Ciascun server ricostruisce, ad intervalli di tempo regolari, a partire dai singoli stati dei nodi, lo stato globale della rete PlanetLab e ne tiene traccia in opportuni file di log. Lo stato di ciascun nodo della rete PlanetLab è definito dai seguenti attributi:

- *FreeSwap:* il totale della memoria di swap libera
- *CPUUser:* percentuale di utilizzo della cpu da parte di processi in userspace

- *CPUSys:* percentuale di utilizzo della cpu da parte del kernel
- *CPUIdle:* Idle time
- *CPUuse:* carico sulla cpu
- *UPtime:* tempo di vita dell'host
- *Load:* numero di processi in esecuzione sul processore negli ultimi 5 minuti
- *MemPress:* percentuale della capacità di allocare memoria
- *MemInfo:* percentuale di memoria libera
- *TXRate:* misura della banda in uscita
- *RXRate:* misura della banda in entrata
- *LiveSlices:* numero di macchine virtuali attive sul nodo

Nelle figure seguenti, dalla 8.1 alla 8.12, sono mostrate le distribuzioni dei valori di ciascun attributo, ognuno nel range in cui è definito. Ad esempio l'attributo *CPUUser* è definito nel range [0;100]. Notiamo come, mentre la distribuzione dei valori per l'attributo *freeSwap* è uniforme, per molti altri attributi i valori si concentrino sulla parte alta o bassa dell'intervallo. Per quanto riguarda l'attributo *uptime*, il grafico è poco significativo poichè l'attributo assume valori crescenti nel tempo ed è molto probabile che non esistano due host con lo stesso valore per quell'attributo.

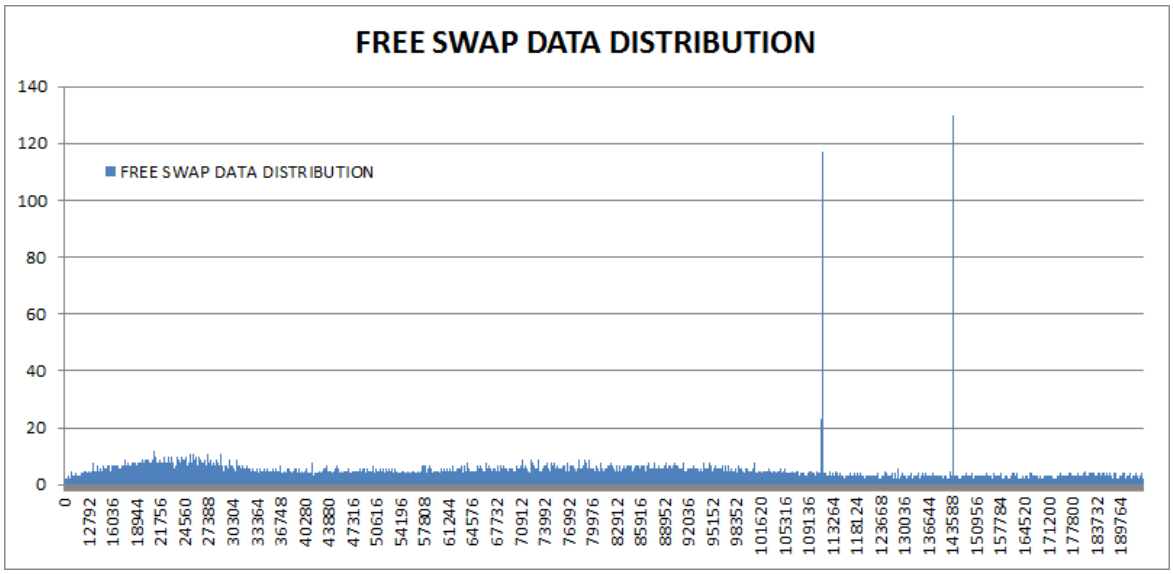


Figura 8.1 – Distribuzione dei valori dell'attributo *freeSwap*

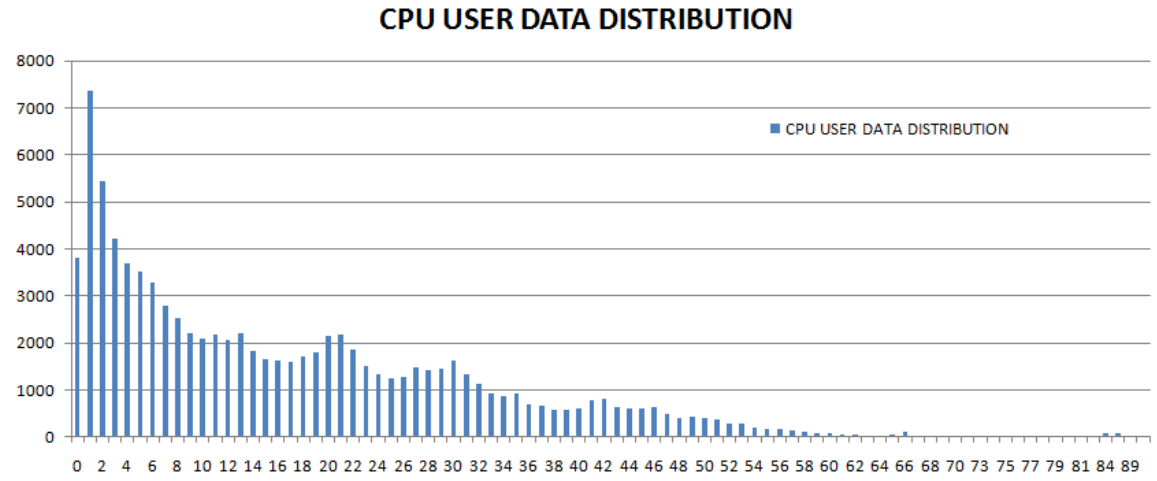


Figura 8.2 – Distribuzione dei valori dell'attributo *cpuUser*

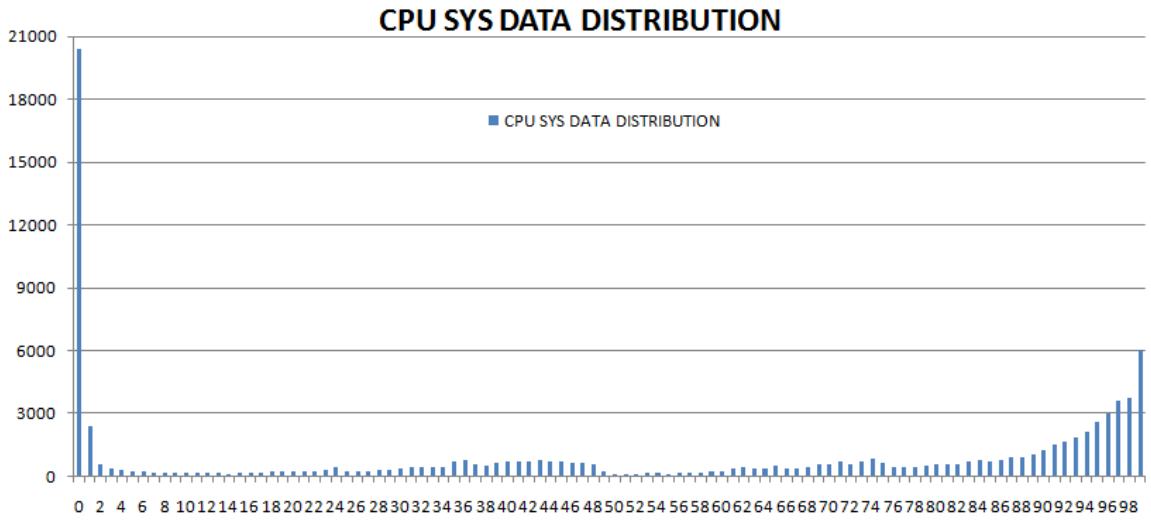


Figura 8.3 – Distribuzione dei valori dell'attributo *cpuSys*

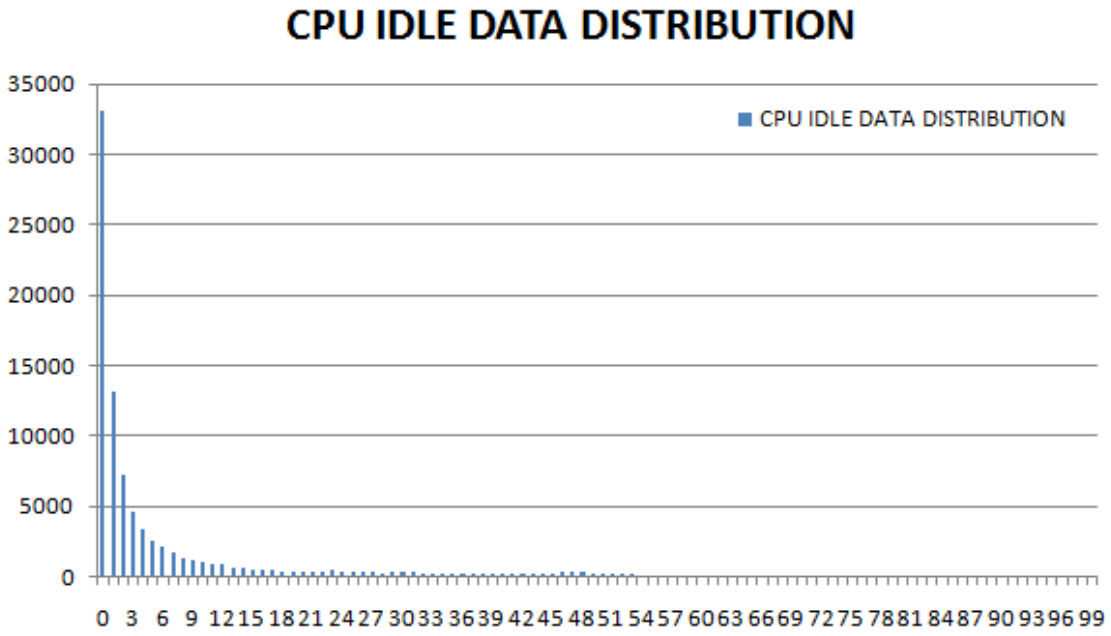


Figura 8.4 – Distribuzione dei valori dell'attributo *cpuIdle*

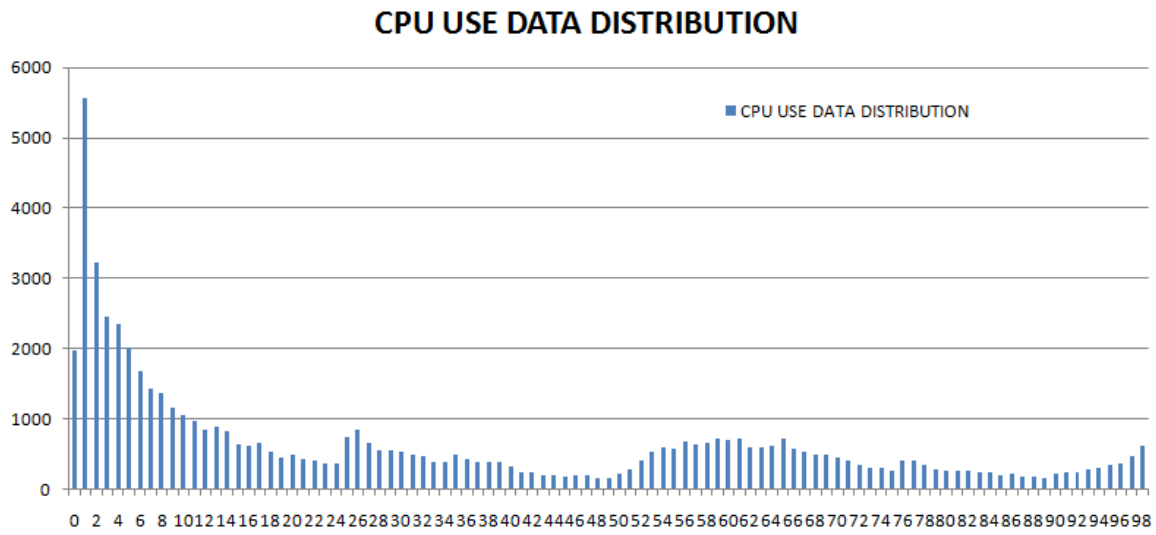


Figura 8.5 – Distribuzione dei valori dell'attributo *cpuUse*

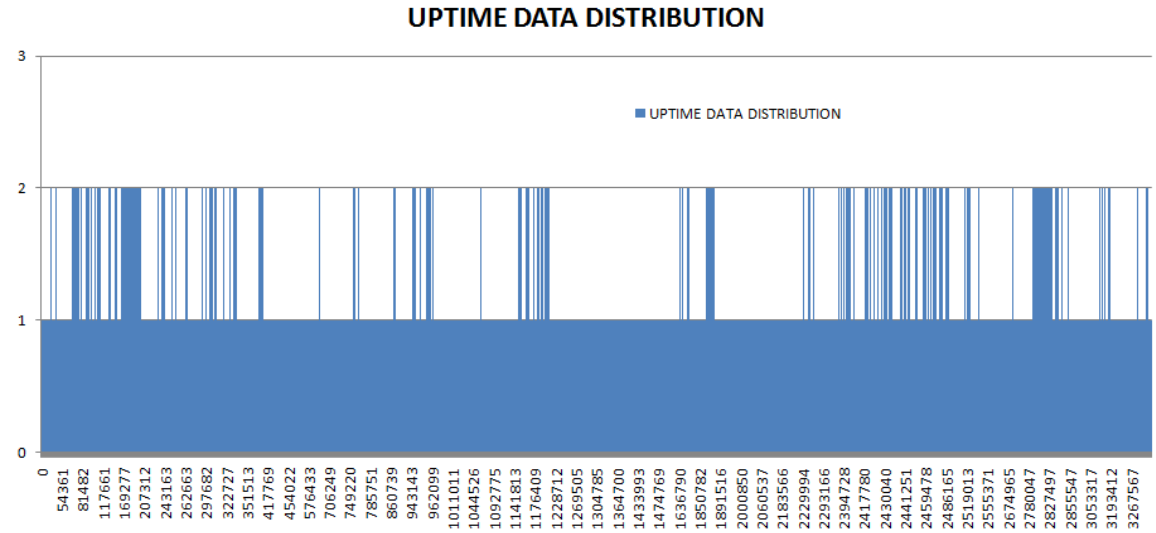


Figura 8.6 – Distribuzione dei valori dell'attributo *uptime*

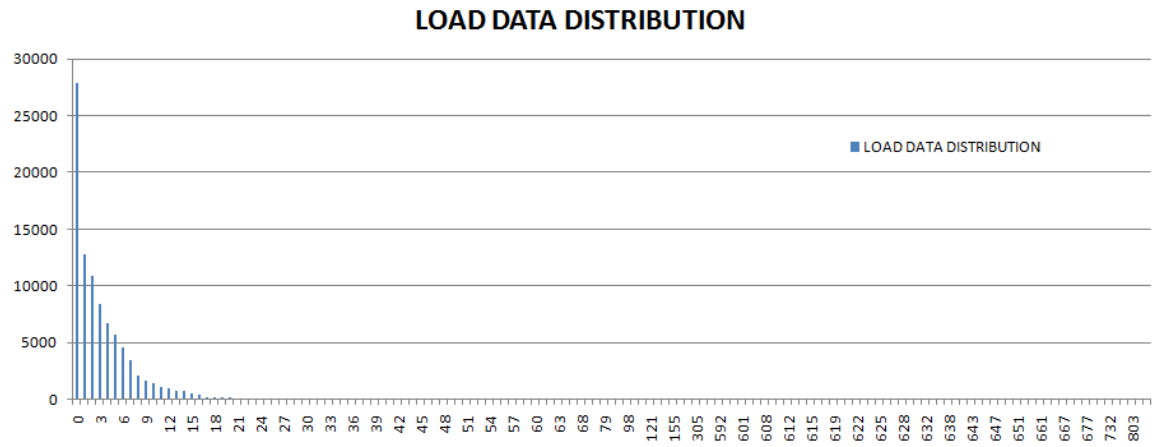


Figura 8.7 – Distribuzione dei valori dell'attributo *load*

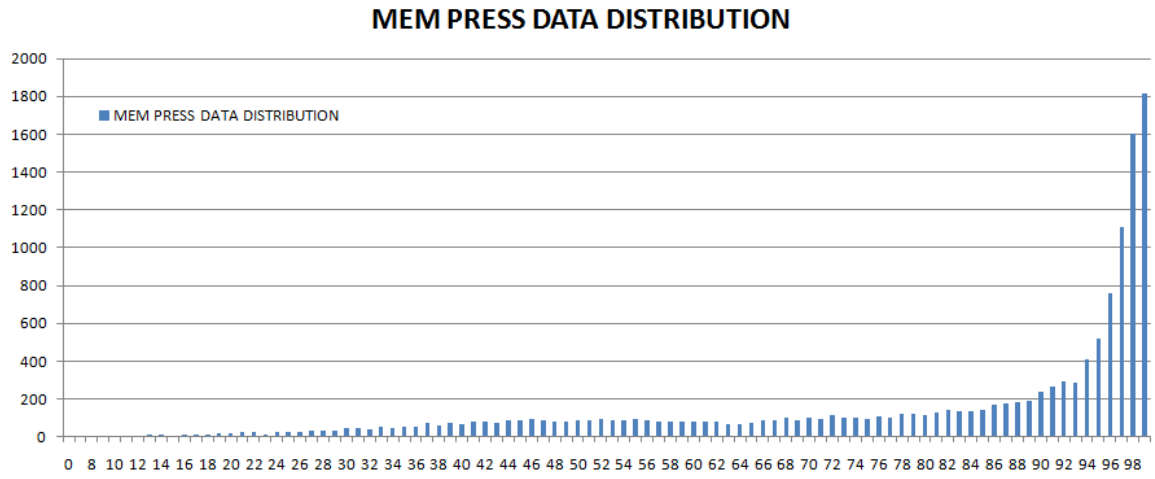


Figura 8.8 – Distribuzione dei valori dell'attributo *memPress*

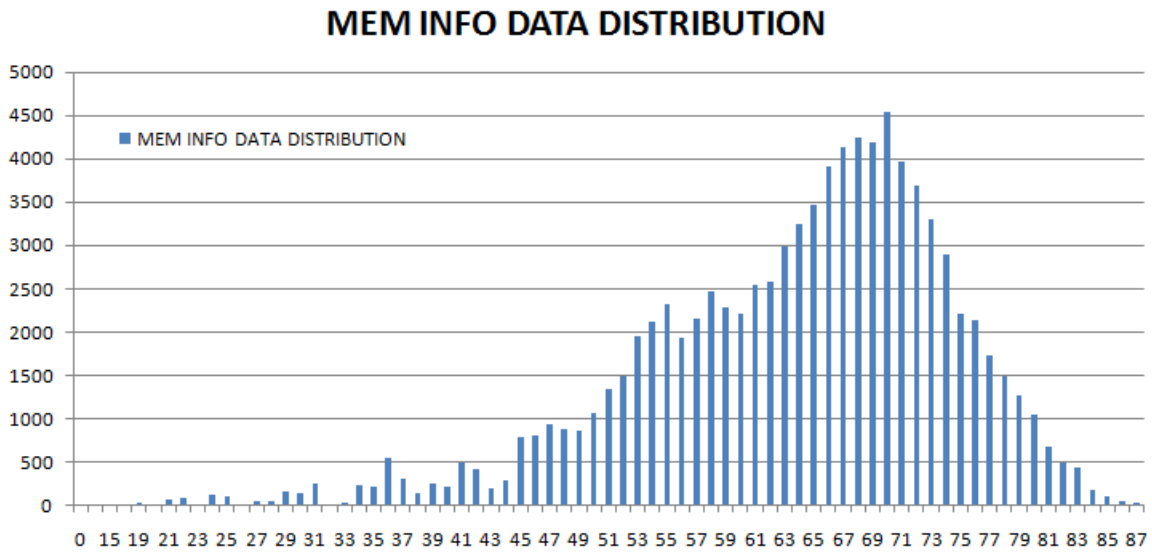


Figura 8.9 – Distribuzione dei valori dell'attributo *memInfo*

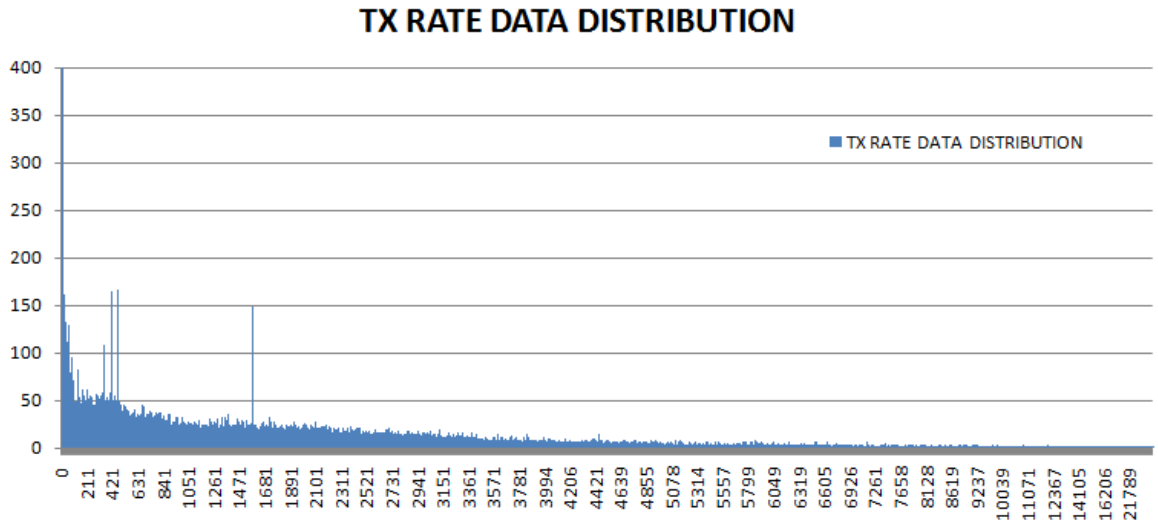


Figura 8.10 – Distribuzione dei valori dell'attributo *TXRate*

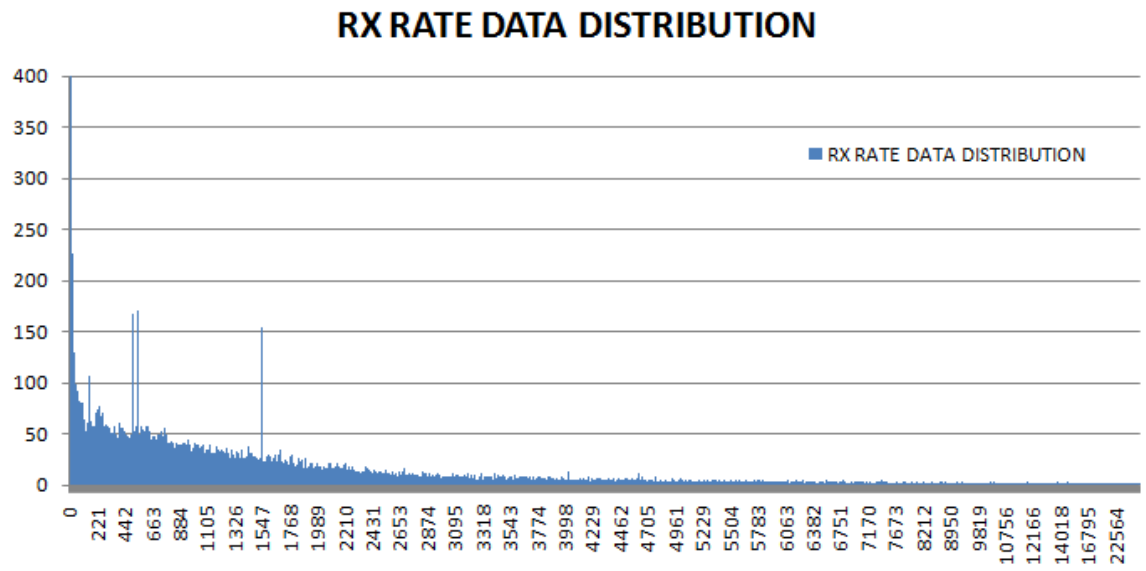


Figura 8.11 – Distribuzione dei valori dell'attributo *RXRate*

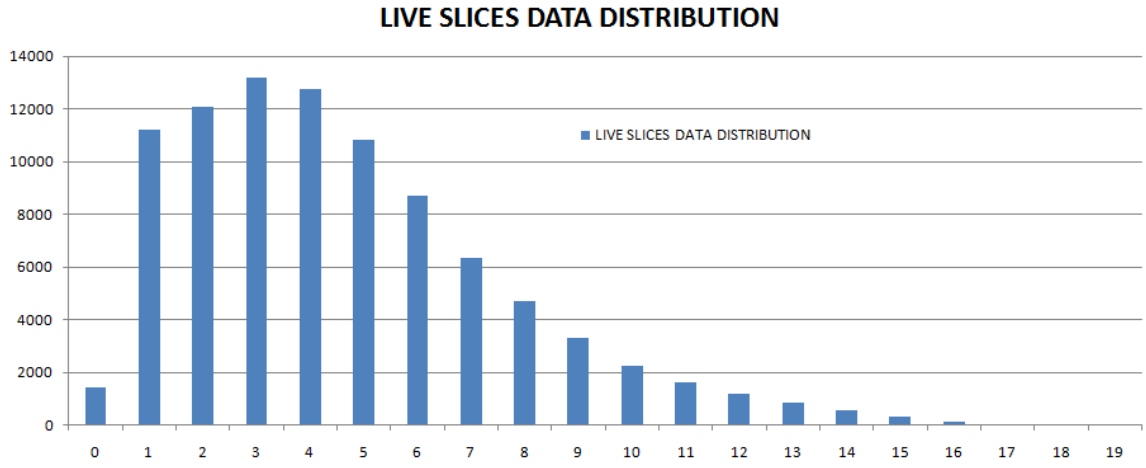


Figura 8.12 – Distribuzione dei valori dell'attributo *liveSlices*

In figura 8.13 è mostrata la distribuzione delle chiavi derivate generate mediante la curva Z-order utilizzando i dati reali di 715 host connessi alla rete PlanetLab al tempo

0 e scegliendo un numero di attributi n=6 e un ordine della curva k=4.

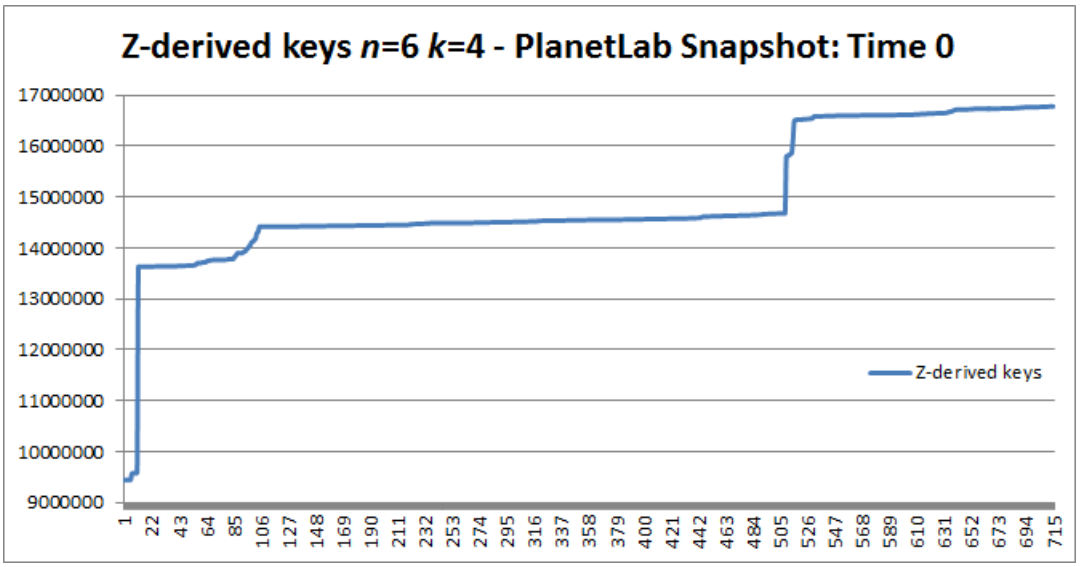


Figura 8.13 – Distribuzione delle chiavi derivate generate mediante la curva Z-order utilizzando i dati reali di host connessi alla rete PlanetLab al tempo 0

8.2 Definizione dei test

I test sono stati eseguiti per ciascuna delle strutture di digest presentate nel capitolo 5, utilizzando i medesimi file di configurazione per quanto riguarda sia il setting dei valori dei vari attributi sia per quanto riguarda la definizione delle range query multiattributo. I vari test sono stati efffettuati in locale tramite l'utilizzo del tool emulatore presente nel framework *OverlayWeaver*. Il numero di nodi virtuali inseriti nella rete *HASP* è 715, pari al numero di host di cui sono disponibili dati reali in PlanetLab. Tale numero indica inoltre il numero di chiavi derivate che verranno memorizzate sulla rete, in quanto nell'implementazione attuale di *HASP*, ciascun nodo memorizza un'unica chiave. Le chiavi derivate sono dunque generate a partire da *log* di che rappresentano lo stato delle macchine connesse alla rete *PlanetLab* in un certo istante di tempo. Lo spazio delle chiavi derivate utilizzate nei test è pari a $[0\,;2^{60}-1]$. Ciascuna chiave derivata è generata a partire dal valore di 6 attributi (dimensioni) ciascuno dei quali può assumere un valore compreso nell'intervallo $[0\,;2^{10}-1]$. Il valore dei parametri n,

numero di dimensioni e k, ordine di ciascun attributo, è pari rispettivamente a 6 e 10 e sono stati scelti in quanto definiscono un possibile scenario tipico. In figura 8.14 è mostrato un frammento del file di configurazione degli attributi utilizzato nei test.

```
schedule 0 control 0 setdynamic freeSwap 93 - cpuUse 1023 - load 9 - memInfo 856 - TXRate 17 - RXRate 29
schedule 1 control 1 setdynamic freeSwap 159 - cpuUse 931 - load 0 - memInfo 726 - TXRate 20 - RXRate 38
schedule 2 control 2 setdynamic freeSwap 49 - cpuUse 266 - load 2 - memInfo 821 - TXRate 100 - RXRate 94
schedule 3 control 3 setdynamic freeSwap 57 - cpuUse 379 - load 8 - memInfo 904 - TXRate 106 - RXRate 108
schedule 4 control 4 setdynamic freeSwap 345 - cpuUse 235 - load 2 - memInfo 797 - TXRate 61 - RXRate 72
schedule 5 control 5 setdynamic freeSwap 32 - cpuUse 1023 - load 5 - memInfo 952 - TXRate 159 - RXRate 197
schedule 6 control 6 setdynamic freeSwap 125 - cpuUse 164 - load 2 - memInfo 845 - TXRate 61 - RXRate 66
schedule 7 control 7 setdynamic freeSwap 144 - cpuUse 1023 - load 11 - memInfo 892 - TXRate 162 - RXRate 185
schedule 8 control 8 setdynamic freeSwap 108 - cpuUse 1023 - load 9 - memInfo 726 - TXRate 75 - RXRate 60
schedule 9 control 9 setdynamic freeSwap 93 - cpuUse 1023 - load 9 - memInfo 904 - TXRate 30 - RXRate 37
schedule 10 control 10 setdynamic freeSwap 91 - cpuUse 215 - load 6 - memInfo 821 - TXRate 258 - RXRate 226
```

Figura 8.14 – Frammento del file di configurazione delle chiavi

Gli attributi utilizzati nei test sono un sottoinsieme di tutti gli attributi analizzati in *PlanetLab*. I valori degli attributi sono stati estratti da file contenenti *snapshot* dello stato dei nodi della rete *PlanetLab* catturati ad intervalli di tempo differenti. I sei attributi scelti sono stati ritenuti i più significativi per la caratterizzazione di uno scenario tipico e sono:

- *freeSwap*
- *cpuUse*
- *load*
- *memInfo*
- *TXRate*
- *RXRate*

Le range query multiattributo sono generate in modo da tenere in considerazione i dati reali stessi, così da adattare l'intervallo di ricerca, per ogni singolo attributo, alla distribuzione dei suoi valori. Per ciascun attributo si proporziona la distribuzione dei 715 valori sull'intervallo $[0\,;2^{10}-1]$ e per ogni valore intero che l'attributo può assumere, se ne calcola la frequenza. In figura 8.15 è riportata la distribuzione dei valori dell'attributo *memInfo* riproporzionata sul range [0;1023].

Una volta calcolate le frequenze dei valori assunti dall'attributo sul range [0;1023] si genera in maniera casuale, per ciascun attributo, un valore *random* compreso nell'intervallo $[0.0\,;1.0]$.

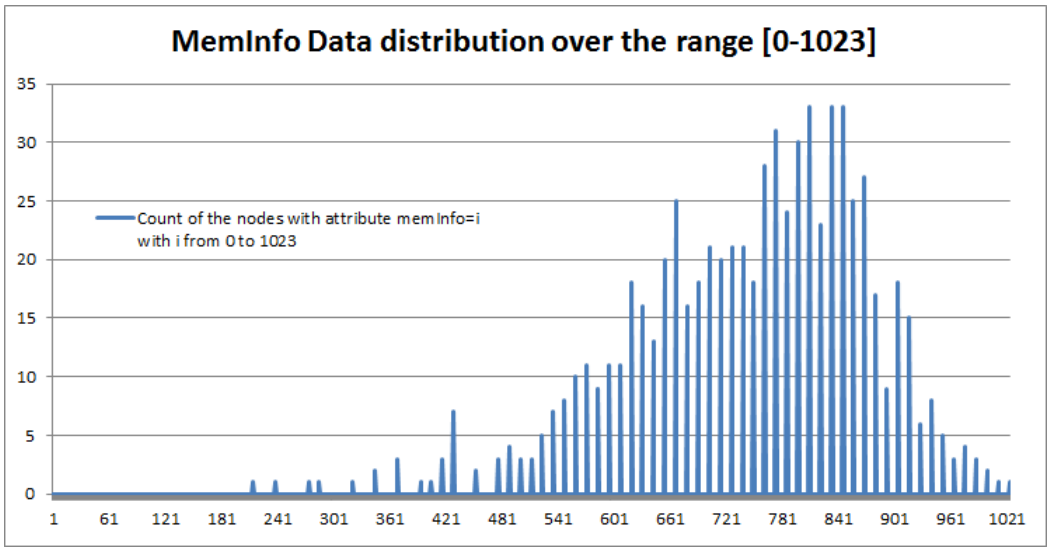


Figura 8.15 – Distribuzione dei valori dell'attributo *memInfo* riproporzionata sul range [0;1023]

Il centro dell'intervallo $[a\,;b]$ che definisce la range query per ogni singolo attributo è dato dal valore x la cui somma delle frequenze di tutti i valori minori uguali di x è pari al valore *random* generato casualmente nell'intervallo $[0.0\,;1.0]$. I valori a e b che definiscono la range query per ogni attributo $[a\,;b]$ sono definiti come:

$$a{=}x{-}range/2$$

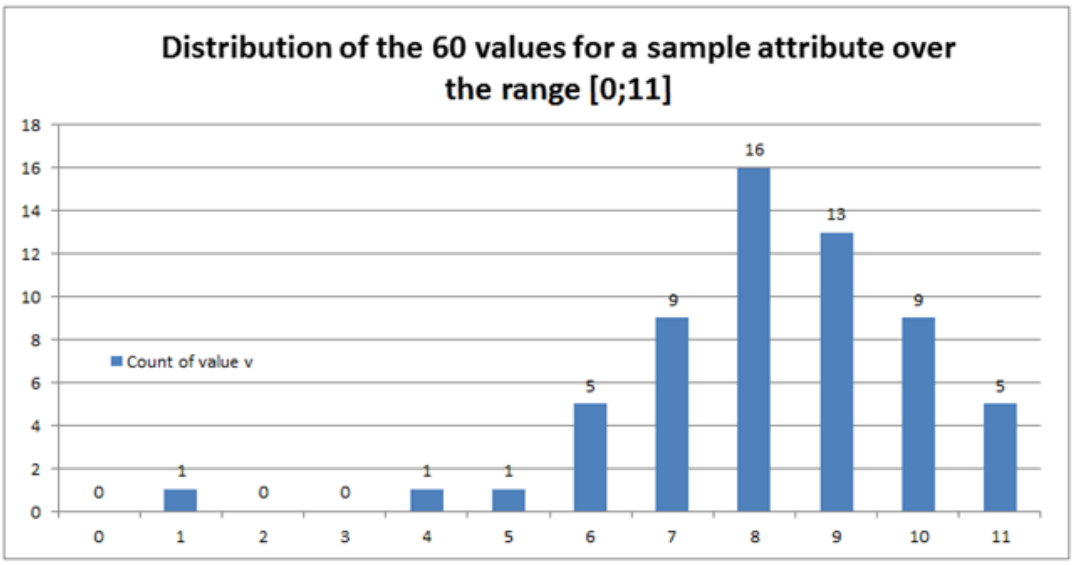
$$b{=}x{+}range/2$$

dove *range* esprime l'ampiezza massima dell'intervallo della range query. Se il valore delle variabili a o b in seguito a tale calcolo risulta rispettivamente minore di zero o maggiore del limite massimo dell'intervallo, $2^k - 1$, il valore della variabile a è settato a zero o il valore della variabile b pari al valore massimo. Il valore della variabile *range* è stabilito in base sia alla dimensione dello spazio dei valori di ciascun attributo sia dal numero di attributi che comporranno la range query multiattributo. Se si assegna un valore basso alla variabile *range,* in caso di range query multiattributo

con un elevato numero di attributi, la range query di ciascun attributo risulterà essere troppo restrittiva e la range query multiattributo non avrà alcuna chiave che la soddisfi. Ricordiamo infatti che le singole condizioni imposte dalle range query dei vari attributi sono tra loro concatenate tramite logica *and*.

Di seguito è riportato un esempio della generazione di una range query guidata dai dati.

Esempio 8.2.1. La range query per ogni singolo attributo, come detto, sono state definite in base alla distribuzione dei dati stessi degli attributi. In questo esempio sarà mostrato il procedimento per la definizione per la range query su un attributo a partire dalla sua distribuzione dei dati. Supponiamo di avere un attributo la cui distribuzione dei valori in un range [0;11] sia nota e mostrata in figura 8.16.



Frequenze:	0/60	1/60	0/60	0/60	1/60	1/60	5/60	9/60	16/60	13/60	9/60	5/60
	0	0.016	0	0	0.016	0.016	0.083	0.15	0.266	0.216	0.15	0.083

Figura 8.16 – Esempio di distribuzione dei 60 valori di un attributo sul range [0;11]

Dalla distribuzione dei 60 valori assunti dall'attributo, è possibile ricavare, per ciascun valore nel range [0;11], la sua frequenza, ovvero il numero di volte che il valore h-esimo

si è presentato, con $h \in [0; 11]$. La somma delle frequenze dei vari valori è pari ad uno.

Valori:	0	1	2	3	4	5	6	7	8	9	10	11
Frequenza:	0/60	1/60	0/60	0/60	1/60	1/60	5/60	9/60	16/60	13/60	9/60	5/60
	0	0.016	0	0	0.016	0.016	0.083	0.15	0.266	0.216	0.15	0.083
Somma delle frequenze precedenti:	0	0.016	0.016	0.016	0.033	0.05	0.133	0.283	0.55	0.766	0,916	1

Figura 8.17 – Frequenze di ciascun valore assunto dall'attributo in figura 8.16

Una volta calcolate le frequenze di ciascun valore appartenente al range [0;11] in cui è definito l'attributo, si genera un valore *random* casuale nell'intervallo [0.0; 1.0]. In figura 8.17 è mostrata la frequenza relativa a ciascun valore che l'attributo nell'esempio può assumere e la somma delle frequenze dei valori precedenti il valore h-esimo. Supponiamo che nel nostro esempio, la variabile *random* assuma un valore pari a 0.6. Il punto centrale della range query è dato dal valore del range [0;11] la cui somma delle frequenze dei valori che lo precedono è pari a 0.6. Nel nostro esempio il punto centrale della range query sarà dunque pari al valore 9 in quanto se alla somma delle frequenze dei valori [0;8], pari a 0.55, si aggiunge la frequenza del valore 9 (0.216) si ottiene un valore 0.766 maggiore uguale al valore *random*. Una volta ricavato il punto centrale x dell'intervallo $[a\,; b]$ della range query, si determinano i valori degli estremi a e b tramite il seguente calcolo:

$$a = x - range/2$$

$$b = x + range/2$$

Nel nostro esempio assegnamo alla variabile *range* un valore pari a 4. La range query che si otterrà per l'attributo in esempio sarà dunque:

7:11

E' possibile affermare che tale range query è stata definita in maniera guidata dalla distribuzione dei valori dell'attributo stesso poichè il punto centrale della range query è stato scelto tenendo conto delle probabilità che ciascun valore appartenente al range [0 ;11] si manifesti per tale attributo.

8.3 Risultati dei test

Le range query multiattributo utilizzate nei test sono dunque composte da $n=6$ intervalli di ricerca, uno per ogni attributo e da una variabile y la quale esprime il numero di risorse cercate che soddisfino contemporaneamente i sei range $[a\,;b]$ stabiliti per ogni attributo.

In figura 8.18 è mostrato un frammento del file di configurazione delle range query multiattributo utilizzato nei test.

```
schedule 0 control 0 getdynamic freeSwap 0:220 cpuUse 526:926 load 0:217 memInfo 383:783 TXRate 0:303 RXRate 0:214 5
timeoffset +10000
schedule 0 control 1 getdynamic freeSwap 0:220 cpuUse 526:926 load 0:217 memInfo 383:783 TXRate 0:303 RXRate 0:214 5
timeoffset +10000
schedule 0 control 2 getdynamic freeSwap 0:220 cpuUse 526:926 load 0:217 memInfo 383:783 TXRate 0:303 RXRate 0:214 5
timeoffset +10000
schedule 0 control 3 getdynamic freeSwap 0:220 cpuUse 526:926 load 0:217 memInfo 383:783 TXRate 0:303 RXRate 0:214 5
timeoffset +10000
schedule 0 control 4 getdynamic freeSwap 0:220 cpuUse 526:926 load 0:217 memInfo 383:783 TXRate 0:303 RXRate 0:214 5
timeoffset +10000
schedule 0 control 5 getdynamic freeSwap 0:220 cpuUse 526:926 load 0:217 memInfo 383:783 TXRate 0:303 RXRate 0:214 5
timeoffset +10000
schedule 0 control 6 getdynamic freeSwap 0:220 cpuUse 526:926 load 0:217 memInfo 383:783 TXRate 0:303 RXRate 0:214 5
```

Figura 8.18 – Frammento del file di configurazione delle range query multiattributo utilizzato nei test.

Per l'esecuzione dei test delle varie strutture di digest sono state definite 10 range query multiattributo differenti e ciascuna di esse è stata eseguita con una strategia *round robin* da ciascuno dei 715 nodi emulati sulla macchina locale. I dati raccolti si basano quindi sull'esecuzione di un totale di 7150 range query. Le dieci range query utilizzate nei test sono riportate in figura 8.19.

```
Range Query 1  --> freeSwap 0:220 cpuUse 526:926 load 0:217 memInfo 383:783 TXRate 0:303 RXRate 0:214 y=5
Range Query 2  --> freeSwap 0:278 cpuUse 342:742 load 0:203 memInfo 454:854 TXRate 0:300 RXRate 0:208 y=5
Range Query 3  --> freeSwap 0:251 cpuUse 0:333 load 0:235 memInfo 312:712 TXRate 0:257 RXRate 0:369 y=5
Range Query 4  --> freeSwap 0:231 cpuUse 823:1023 load 0:203 memInfo 466:866 TXRate 0:200 RXRate 0:207 y=5
Range Query 5  --> freeSwap 0:236 cpuUse 0:323 load 0:200 memInfo 585:985 TXRate 0:357 RXRate 0:225 y=5

Range Query 6  --> freeSwap 0:216 cpuUse 383:783 load 0:224 memInfo 645:1023 TXRate 0:336 RXRate 0:340 y=5
Range Query 7  --> freeSwap 0:232 cpuUse 823:1023 load 0:200 memInfo 538:938 TXRate 0:212 RXRate 0:314 y=5
Range Query 8  --> freeSwap 61:461 cpuUse 66:466 load 0:202 memInfo 454:854 TXRate 0:237 RXRate 0:341 y=5
Range Query 9  --> freeSwap 0:340 cpuUse 526:926 load 0:200 memInfo 573:973 TXRate 0:274 RXRate 0:293 y=5
Range Query 10 --> freeSwap 6:406 cpuUse 312:712 load 0:202 memInfo 668:1023 TXRate 0:237 RXRate 0:252 y=5
```

Figura 8.19 – Range queries multiattributo utilizzate per i test

Le misure analizzate nei test sono:

1. sovrastima media del numero di risultati restituiti dal processo di risoluzione di una query
2. sbilanciamento medio del numero di nodi contattati: differenza tra il numero massimo e minimo di nodi contattati nella risoluzione della stessa range query eseguita da nodi diversi
3. percentuale media di query che raggiungono la radice dell'albero *HASP* nel processo di risoluzione di una query
4. percentuale media del numero di nodi contattati rispetto al numero di nodi totali della rete durante il processo di risoluzione di una query

Il primo test fornisce una misura della *query satisfaction*, ovvero del numero medio di *match* restituiti per ciascuna query di test. E' un dato di fatto che un'informazione di digest può sottostimare il numero di match di una query che sono presenti in un sottoalbero. Quando ciò accade, il numero di risultati restituiti al nodo che ha effettuato la query è maggiore del numero k richiesto.

Il numero di nodi coinvolti nel processo di risoluzione di una query può variare a seconda di quale nodo dell'albero invochi la query. Il secondo test misura lo sbilanciamento medio del numero di nodi contattati nell'esecuzione della stessa query da parte di nodi diversi.

Uno dei punti cruciali della proposta *HASP* è l'impatto del traffico generato dal processo di risoluzione della query sul peer che gestisce il nodo radice dell'albero. Il terzo test si propone di investigare questo aspetto e per questo si analizza il numero medio di query che raggiungono il peer che gestisce la radice dell'albero di aggregazione. Un'informazione di digest è utile per riassumere in un nodo l'informazione contenuta nel sotto-albero radicato in tale nodo. Una strategia di aggregazione è tanto migliore quanto maggiore è l'accuratezza delle informazioni contenute nei nodi riguardo il suo sotto-albero. Tanto minore è il numero di nodi contattati rispetto al numero totale di nodi, durante il processo di risoluzione di una query, tanto migliore è la strategia di aggregazione utilizzata.

Il grafico in figura 8.20 mostra la sovrastima media, espressa in percentuale, del numero di risorse trovate da ciascuna query rispetto al valore k di risorse richieste.

Ricordiamo che quando si parla di valore medio per ciascuna query, si intende il valore medio calcolato dall'esecuzione della stessa query da parte dei 715 nodi della rete di test. E' possibile notare come per 6 query su 10, la percentuale media di sovrastima dei risultati richiesti rispetto a quelli cercati sia sensibilmente minore, con una media del 15%, utilizzando la strategia di aggregazione *data-driven BitVector (dBV)* rispetto all'utilizzo dell'altra strategia *BitVector* con *intervalli equidistanti* non guidati dai dati (*eBV*). Ad esempio, nel caso della query numero 4, si ha una sovrastima minore del 22% con la strategia dBV rispetto a quella eBV, ad indicare come la prima strategia fornisca un'approssimazione più precisa delle informazioni contenute nei sottoalberi dei nodi in *HASP.* Nelle quattro query rimanenti, il *data-driven BitVector* si comporta all'incirca come l'altra strategia di aggregazione eBV con differenze poco significative dell'ordine dell'1%.

Il grafico in figura 8.21 mostra lo sbilanciamento medio, espresso in percentuale, tra il massimo ed il minimo numero di nodi contattati durante il processo di risoluzione di una query eseguita da parte di nodi differenti. E' possibile notare come per 8 query su 10, lo sbilanciamento medio del numero di nodi contattati da ciascuna query sia minore, con una media del 3.4%, utilizzando la strategia di aggregazione *data-driven BitVector (dBV)* rispetto all'utilizzo dell'altra strategia *BitVector* con *intervalli equidistanti* non guidati dai dati (*eBV*). Ad esempio, nel caso della query numero 4, si ha una uno sbilanciamento tra il numero massimo e minimo di nodi contattati minore dell' 8.2% con la strategia dBV rispetto a quella eBV. Nelle altre due query, la tecnica data-driven BitVector si comporta all'incirca come l'altra strategia di aggregazione eBV con differenze poco significative dell'ordine dell'1%.

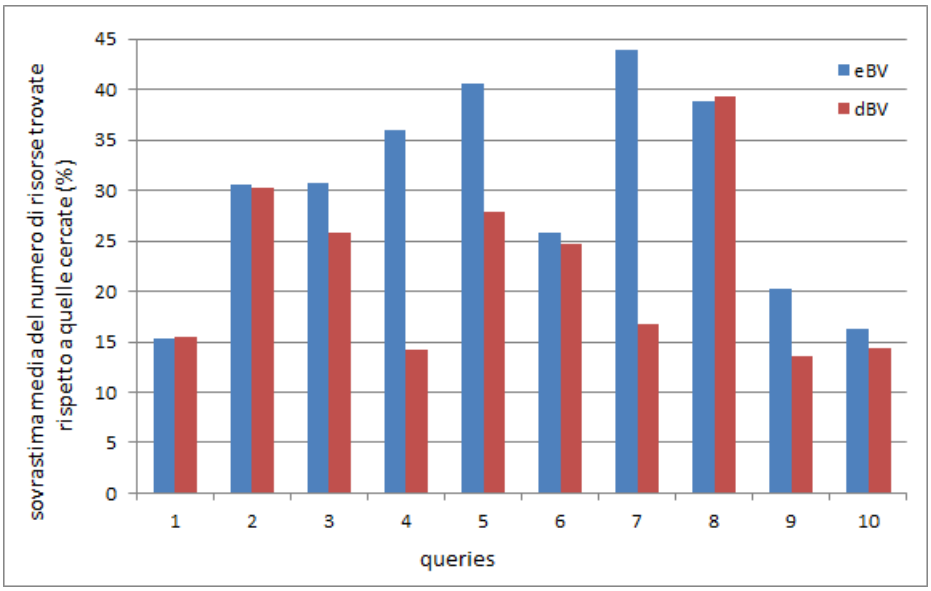


Figura 8.20 – *Test 1:* sovrastima media in percentuale del numero di risultati restituiti rispetto al valore k richiesto

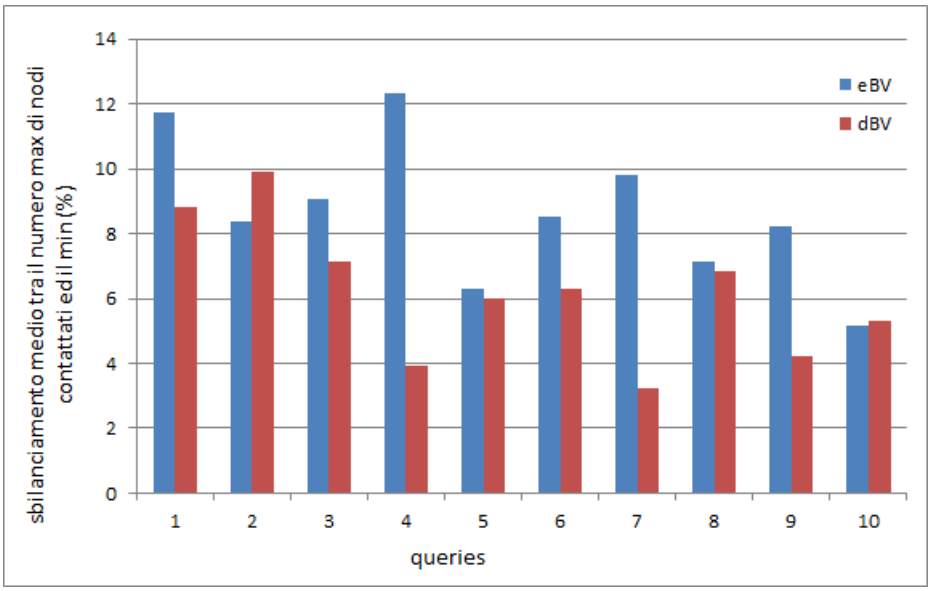


Figura 8.21 – *Test 2:* sbilanciamento medio del numero di nodi contattati nel processo di risoluzione della query

Il grafico in figura 8.22 mostra il valore medio, espresso in percentuale, del numero

di query che durante il processo di risoluzione di una query eseguita da parte di nodi differenti raggiungono la radice dell'albero di *HASP*. E' possibile notare come per tutte e 10 le query in esame, le due strategie di aggregazione *data-driven BitVector (dBV)* e *BitVector* con *intervalli equidistanti* (*eBV*) ottengano i medesimi risultati.

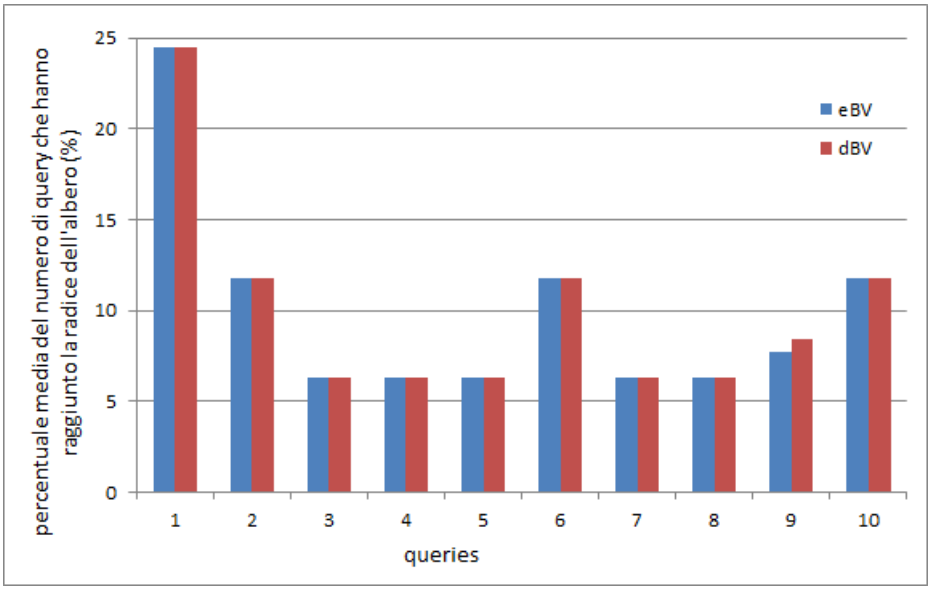


Figura 8.22 – *Test 3:* percentuale media del numero di query che raggiungono la radice dell'albero *HASP*

Il grafico in figura 8.23 mostra il numero medio di nodi contattati rispetto al numero di nodi totali della rete, espresso in percentuale, durante il processo di risoluzione di una query eseguita da parte di nodi differenti. E' possibile notare come per 8 query su 10, il numero medio di nodi contattati rispetto al numero di nodi totali della rete sia minore, con una media di poco superiore all'1%, utilizzando la strategia di aggregazione *data-driven BitVector (dBV)* rispetto all'utilizzo dell'altra strategia *BitVector* con *intervalli equidistanti* non guidati dai dati (*eBV*). Nel caso della query numero 1, il numero medio di nodi contattati rispetto al numero di nodi totali è minore di circa il 3% con la strategia dBV rispetto a quella eBV. Nelle altre due query, il *data-driven BitVector* si comporta all'incirca come l'altra strategia di aggregazione eBV con differenze poco significative minori dell'1%.

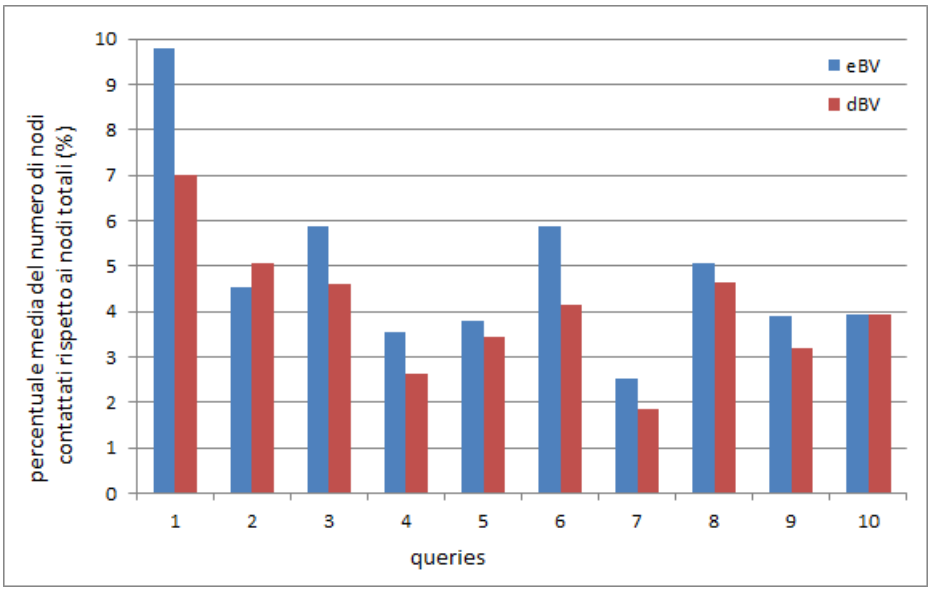


Figura 8.23 – *Test 4:* percentuale media del numero di nodi contattati rispetto al numero di nodi totali

Capitolo 9

Conclusioni

Questa tesi ha definito un supporto per la gestione efficace di operazioni di ricerca basate su range query multiattributo in sistemi P2P. Il sistema proposto, *HASP*, è basato su una struttura di tipo gerarchico progettata in maniera distribuita e tale da permettere, in seguito, l'utilizzo di diverse strategie di mapping e di aggregazione delle informazioni, ma anche di curve space-filling diverse rispetto alla curva *Z-order*. La scelta della curva space-filling Z-order è basata su un compromesso tra il grado di clustering [35], minore rispetto ad altre curve, e la semplicità di definizione degli algoritmi di generazione della chiave derivata e del processo di risoluzione di range query multidimensionali. La complessità degli algoritmi di generazione della chiave derivata è $O(n \cdot k)$ sia per la curva di Hilbert che per la curva Z-order, con n numero di dimensioni e k ordine di raffinamento della curva space-filling, ma la complessità dell'implementazione dei due algoritmi si è rivelata tuttavia molto differente. L'approccio seguito per la risoluzione di una query multiattributo è basato su una soluzione guidata dai dati in modo da evitare la generazione di tutti i segmenti della curva space-filling contenuti nell'iper-rettangolo che definisce la query e di considerare i soli segmenti che contengono informazioni significative. L'elevata dimensione dello spazio dei valori delle chiavi derivate ha richiesto nuove implementazioni delle strategie di digest presenti in XCone [44] ed in particolare è stata necessaria la definizione e l'utilizzo di una struttura dati dinamica per la strategia *q-digest*. I test effettuati su un *dataset reale* hanno evidenziato come l'introduzione delle tecniche di digest abbia apportato un reale guadagno delle prestazioni nell'esplorazione della struttura.

9.1 Sviluppi futuri

Nell'attuale versione di *HASP* si utilizza la curva space-filling Z-order per la generazione della chiave derivata. Dallo studio delle proprietà di clustering delle varie curve space-filling si è potuto notare come vi siano curve di questo tipo che preservano un grado maggiore di località dei dati. In futuro potrebbero essere integrate in *HASP* nuove curve space-filling con carattteristiche diverse, semplicemente inserendo i nuovi algoritmi per la generazione della chiave derivata nel modulo *KeyManager*, grazie all'elevata modularità del sistema.

HASP, attualmente, utilizza la *funzione di aggregazione massimo* come *funzione di mapping* dei nodi logici dell'albero ai peer della rete. Uno dei possibili sviluppi consiste nell'analizzare nuove funzioni di mapping ed in tale funzioni tener conto di altri fattori come la banda di connessione dei vari nodi, la potenza di calcolo o il carico di nodi logici gestiti.

Le funzioni di digest presenti nella versione attuale di *HASP,* le due varianti per il *BitVector* ed il *q-digest*, possono essere affiancate da ulteriori funzioni di aggregazione che migliorino l'approssimazione delle informazioni contenute nei digest dei vari nodi, al fine di rendere più efficiente il processo di esplorazione dell'albero *HASP*.

Nel processo di risoluzione di una range query, nella versione attuale *di HASP*, a partire dalle informazioni di digest si sceglie un segmento di curva space-filling per calcolarne l'intersezione con la range query, senza tener conto in alcun modo del numero di risorse che esso contiene. Uno sviluppo del sistema potrebbe integrare una o più *euristiche* per determinare i segmenti di curva più *promettenti* tra quelli presenti nell'informazione di digest.

Ringraziamenti

La presente tesi rappresenta non solo la fine di un lungo cammino di studi, ma racchiude le esperienze maturate in questi anni di università e si propone come punto di partenza per un nuovo capitolo della mia vita. Voglio ringraziare i miei genitori per avermi permesso di compiere tale cammino e per avermi spronato nei momenti difficili e la mia fidanzata, Roberta, per avermi sempre sostenuto e sopportato durante questi anni universitari. Un grazie va anche a tutti i miei amici che con il loro appoggio, le loro battute e i loro consigli, mi hanno sempre accompagnato durante questo percorso. Per ultimi, ma non per ordine d'importanza, voglio ringraziare la professoressa Laura Ricci e il professor Massimo Coppola per il supporto e la grande disponibilità dimostrata durante lo svolgimento del presente lavoro di tesi.

Elenco delle figure

.

Bibliografia

[1] R.Steinmetz and K. Wehrle. *Peer to Peer Systems and Applications.* LNCS. 3485, Springer Verlag, 2005.

[2] R. Matei, A. Iamnitchi, and P. Foster. *Mapping the gnutella network.* Internet Computing, IEEE, 6(1):50-57, 2002.

[3] *Napster.* http://www.napster.com/, 2006.

[4] J. A. Pouwelse, P. Garbacki, D. H. Epema, and H. J. Sips. *The bittorrent P2P file-sharing system: Measurements and analysis.* Proceedings of the 54th International Workshop on Peer-to-Peer Systems (IPTPS'05), pages 205-216. 2005, Ithaca, USA, 2005.

[5] Li Gong. *JXTA: a network programming environment.* Internet Computing, IEEE, 5(3):88-95, 2001.

[6] N. Leibowitz, M. Ripeanu, and A. Wierzbicki. *Deconstructing the Kazaa network.* Proceedings of the The Third IEEE Workshop on Internet Applications (WIAPP '03), pages 112-120, June 2003.

[7] C. Kesselman and I. Foster. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, November 1998.

[8] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, F. M. Kaashoek, F. Dabek, and H. Balakrishnan. *CHORD: a scalable peer-to-peer lookup protocol for internet applications.* IEEE/ACM Transactions on Networking (TON), 11(1):17-32, February 2003.

[9] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. *A scalable content addressable network.* Technical Report TR00-010, Berkeley, CA, 2000.

[10] A. Rowstron and P. Druschel. *Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems.* Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms, Heidelberg, 2001.

[11] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. *Tapestry: a resilient global-scale overlay for service deployment.* Selected Areas in Communications, IEEE Journal on, 22(1):41-53, 2004.

[12] C. Buragohain, D. Agrawal, and S. Suri. *A game theoretic framework for incentives in p2p systems.* Peer-to-Peer Computing, 2003. (P2P 2003). Proceedings for the Third International Conference on, pages 48-56, September 2003.

[13] I. Foster and A. Iamnitchi. *On death, taxes, and the convergence of peer-to-peer and grid computing.* nd International Workshop on Peer-to-Peer Systems (IPTPS'03), pages 118-128, Berkeley - USA, February 2003.

[14] Petar Maymounkov and David Mazieres. *Kademlia: A Peer-to-Peer Information System Based on the XOR Metric.* Electronic Proceedings for the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02), MIT Faculty Club - Cambridge, March 2002.

[15] M. Cai, M. Frank, J. Chen and P. Szekely. *MAAN: a multi-attribute addressable network for grid information services.* Grid Computing, 2003. Proceedings for the Fourth International Workshop on, pp. 184-191, November 2003.

[16] C. Schmidt and M. Parashar. *Analyzing the search characteristics of space-filling curve-based indexing within the squid p2p data discovery system.* Technical report Vol. TR-276, Rutgers University, December 2004.

[17] H. V. Jagadish, Beng C. Ooi, and Quang H. Vu. *Baton: a balanced tree structure for peer-to-peer networks.* In Proceedings of the 31st Internationational Conference on Very Large Data Bases (VLDB '05), pp. 661-672. Trondheim - Norway, September 2005.

[18] Jun Gao. *A Distributed and Scalable Peer-to-Peer Content Discovery System Supporting Complex Queries.* PhD thesis, Computer Science, Carnegie Mellon University, October 2004.

[19] G. Varghese, R. Bhagwan, P. Mahadevan and G. M. Voelker. *Cone: A distributed heap-based approach to resource selection.* Technical report, University of California, 2004.

[20] D. Carfì. *XCone: Range query in sistemi P2P.* Tesi di Laurea Specialistica, Università di Pisa, Dicembre 2008.

[21] P. Crescenzi, G. Gambosi e R. Grossi. *Strutture di dati e algoritmi.* Pearson - Addison Wesley, 2006

[22] J. K. Lawder. *The Application of Space-filling Curves to the Storage and Retrieval of Multi-dimensional Data.* PhD Thesis, Birkbeck College - University of London, 1999.

[23] T. Bially. *A Class of Dimensions Changing Mappings and its Application to Bandwith Compression.* PhD Thesis - Polytechnic Institute of Brooklin, 1967.

[24] T. Skopal, M. Kratky, J. Pokorny and V. Snasel. *A new range query algorithm for Universal B-trees.* Information Systems Journal Vol. 31 Issue 6, pp. 489-511, September 2006.

[25] B. Mandelbrot. *Gli oggetti frattali*, Torino, Einaudi Editore, 2000.

[26] J.E. Hutchinson. *Fractals and self similarity.* Indiana Univ. Math. J. 30: 713–747, 1981.

[27] P. Prusinkiewicz, J. Hanan, A. Lindenmayer, F.D. Fracchia, K. Krithivasan. *Lindenmayer Systems, Fractals, and Plants.* Springer-Verlag, 1989.

[28] S. Addea. *Risoluzione di range query multiattributo in sistemi P2P: un approccio basato su curve space-filling.* Tesi di Laurea Specialistica, Università di Pisa, Dicembre 2009.

[29] H. Sagan. *Space-Filling Curves.* Universitext series. Springer-Verlag, Heidelberg, 1994.

[30] G. M. Morton. *A computer Oriented Geodetic Data Base and a New Technique in File Sequencing*, Technical Report, Ottawa, 1966.

[31] E. H. Moore. *On certain crinkly curves.* Transactions of the American Mathematical Society, 1:72-90, January 1900.

[32] I. H. Witten and B. Wyvill. *On the generation and use of space-filling curves.* Software - Practice and Experience, 13(6):51 9-525, June 1983.

[33] C. Faloutsos. *Multiattribute hashing using gray codes.* Proceedings of the 1986 ACM SIGMOND International Conference on Management of Data, pp. 227-238 ACM Press, Washington D.C., May 28-30, 1986.

[34] F. Gray. *Pulse code communications.* U.S. Patent 2 632 058, March 1953.

[35] B. Moon, H.V. Jagadish, C. Faloutsos and J. H. Saltz. *Analysis of the Clustering Properties of the Hilbert Space-Filling Curve.* IEEE Transactions on knowledge and data engineering. Vol. 13, No 1, January/February 2001.

[36] H.V. Jagadish. *Linear Clustering of Objects with Multiple Attributes.* Proceedings of the ACM SIGMOND Conference, pp. 332-342, May 1990.

[37] Y. Rong and C. Faloutsos. *Analisys of the Clutering Property of the Peano Curves.* Technical Report CS-TR-2792, UMIACS-TR-91-151, University of Maryland, December 1991.

[38] H.V. Jagadish. *Analisys of the Hilbert Curve for representing Two-Dimensional Space.* Information Processing Letters, Vol. 62, No 1, pp. 17-22, April 1997.

[39] H. Haverkort and F. van Walderveen. *Locality and Bounding Box Quality of Two-Dimensional Space Filling Curves.* Dept. of Computer Science, Eindhoven University of Technology. In Computational Geometry: Theory and Applications. Volume 43, Issue 2, pp. 131-147, February 2010.

[40] N. Shrivastava, C. Buragohain, D. Agrawal and S. Suri. *Medians and beyond: new aggregation techniques for sensor networks.* In SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems, pp. 239-249, New York, NY, USA, 2004. ACM.

[41] K. Shudo, Y. Tanaka, and S. Sekiguchi. *Overlay Weaver: An overlay construction toolkit.* Computer Communications, (2):402-412, February. Framework available at http://overlayweaver.sourceforge.net

[42] F. Dabek, B. Zhao, P. Druschel, and I. Stoica. *Towards a common API for structured peer-to-peer overlays*. Peer-to-Peer Systems II, Second International Workshop, (IPTPS 2003), Berkeley - USA, February 2003.

[43] *PlanetLab: An open platform for developing, deploying, and accessing planetary-scale services*. Website: http://www.planet-lab.org/

[44] D. Carfì, M. Coppola, D. Laforenza and L. Ricci. *DDT: A Distributed Data Structure for the Support of P2P Range Query*. Proceedings of the IEEE CollaborateCom 2009, 5th International Conference on Collaborative Computing, Networking and Applications, Washington, November, 2009.