Università degli studi
di Modena e Reggio Emilia

Facoltà di Scienze
Matematiche, Fisiche e Naturali
Corso di Laurea Magistrale
in Informatica

Gestione Efficiente dei Dati in un Data Stream Management System

Federico Di Bernardo

Tesi di Laurea

Relatore:
Ing. Riccardo Martoglia

Correlatori:
Dott. Federica Mandreoli
Ing. Luca Carafoli

Anno Accademico 2009/2010

Ringraziamenti

Desidero ringraziare l'Ing. Riccardo Martgolia, la Dott. Federica Mandreoli e l'Ing. Luca Carafoli per la grande disponibilità dimostrata, la costante assistenza, la preziosa collaborazione e la fiducia che mi hanno offerto in questi mesi di lavoro.
Ringrazio in particolare mio padre Ennio per avermi sostenuto non solo economicamente ma anche moralmente nei momenti di difficoltà.
Grazie infine ai miei amici e compagni di studi, che tanto mi hanno dato in questi anni di studio passati insieme.

Parole chiave

DSMS
Data Stream
Query continue
Storage Manager
PEGASUS

Indice

Introduzione **1**

I Stato dell'arte **4**

1 Progetto PEGASUS **6**
1.1 Motivazioni e obiettivi del progetto 6
1.2 Descrizione generale del progetto 7
1.2.1 OBU 9
1.2.2 Centro di controllo 11
1.2.3 Comunicazioni V2V 12
1.2.4 Comunicazioni V2I 13
1.3 Centro di controllo 14
1.3.1 Descrizione architettura 14
1.3.2 Service Module 15
1.3.3 Data Stream Management System (DSMS) 15
1.3.4 Gestore delle comunicazioni 22
1.3.5 Recommender System 22

2 Data Stream Management System (DSMS) **26**
2.1 Caratteristiche generali 26
2.1.1 DBMS vs DSMS 26
2.1.2 Data stream 30
2.1.3 Query continue 34
2.2 Scenari di applicazione 37
2.2.1 Reti di sensori 37
2.2.2 Analisi del traffico di rete 42
2.2.3 Analisi del traffico urbano 44
2.2.4 Analisi finanziarie 47
2.2.5 Analisi dei log 49
2.3 Panoramica sui sistemi attuali 51
2.3.1 Aurora 51
2.3.2 TelegraphCQ 56

2.3.3 STREAM 59
2.3.4 NiagaraCQ 62
2.3.5 Cougar 63
2.3.6 Linear Road 65

II Gestione efficiente dei dati in un DSMS 70

3 Architettura di un DSMS efficiente e scalabile 72
3.1 Descrizione generale dell'architettura 72
3.2 Store Management System (SMS) 74
3.2.1 Role e Access Model 74
3.2.2 Sincronizzazione, Schema e Persistenza 76
3.2.3 Parametri Performance-Related 76
3.2.4 Update pattern 78
3.2.5 Read pattern 83
3.2.6 Sharing e Access Pattern 84
3.2.7 Strutture dati 85
3.2.8 Indici 89
3.3 Query Processing Engine (QPE) 91
3.3.1 Semantica e sintassi delle query continue 91
3.3.2 Linguaggi per le query continue 98
3.3.3 Operatori per il data streaming 101
3.3.4 Ottimizzazioni e performance delle query continue 107

4 Prototipo software sviluppato 111
4.1 Scelte tecnologiche e programmative effettuate 111
4.2 Descrizione generale del prototipo 112
4.3 Requisiti funzionali 116
4.3.1 RF01: Lettura dei dati provenienti dalle OBU 116
4.3.2 RF02: Adattamento dinamico delle strutture dati 117
4.3.3 RF03: Gestione di più finestre temporali su dati condivisi . 118
4.3.4 RF04: Migrazione dei dati storici o consumati in memoria secondaria 120
4.3.5 RF05: Esecuzione parallela delle query e restituzione dei risultati 120
4.4 Use Case Diagram 121
4.4.1 Use Case 1: Esecuzione del produttore 121
4.4.2 Use Case 2: Sottomissione di una query consumatore . . . 123
4.5 Moduli e macro funzionalità 126
4.5.1 Grafo delle classi 126
4.5.2 Classe DSMSCoordinator.java 126
4.5.3 Classe StorageManager.java 128

4.5.4 Classe QueryEngine.java 131
4.5.5 Altre classi 132
4.5.6 Interfaccia grafica 134

5 Prove sperimentali e conclusioni 137
5.1 Dataset utilizzato 138
5.2 Analisi delle performance e della scalabilità 140
5.2.1 Stress Testing 140
5.2.2 Load Testing 140
5.2.3 Performance Testing 145
5.3 Conclusioni e sviluppi futuri 146

Elenco delle figure

1.1 Infrastruttura progetto PEGASUS 8
1.2 Architettura On-Board Unit 9
1.3 Architettura centro di controllo del progetto PEGASUS 14
1.4 Interazione tra Store Manager e Query Processing Engine 16
1.5 Il data stream in arrivo dalle OBU viene inserito una finestra temporale su cui verranno eseguite le query. 18
1.6 Un Recommender System elabora i dati sugli utenti C e gli oggetti I che raccomanda, per generare insieme di oggetti personalizzati per ciascun utente 23

2.1 Principio di funzionamento di un DBMS 27
2.2 Principio di funzionamento di un DSMS 28
2.3 Esempio di sistema di calcolo per il data stream. 33
2.4 Il sensore MICA sviluppato alla Berkeley University. 40
2.5 Data stream management system per l'analisi del traffico di rete. . 43
2.6 Architettura del sistema AURORA. 52
2.7 GUI del sistema Aurora con un grafo di esecuzione: (a) a sinistra si notano gli stream sorgenti, al centro il grafo composto dai vari operatori ed a destra gli stream di uscita disponibili per le applicazioni client; (b) la fase di monitoraggio delle code implementate da un particolare grafo di esecuzione. 53
2.8 Aurora query model. 54
2.9 Grafici relativi al QoS. 55
2.10 Architettura TelegraphCQ. 58
2.11 Interfaccia STREAM. 60
2.12 Definizione di monitor di attività su un elemento del grafo di esecuzione di una query. A sinistra viene mostrata la selezione del collegamento da monitorare. A destra, viene creato un monitor che riporta in un diagramma temporale i valori che attraversano il collegamento. 61
2.13 Query plan del nodo sorgente e del nodo leader. 64
2.14 Linear City. 68
2.15 Struttura di una expressway. 69

3.1 A sinistra, un DSMS in cui la componente di gestione della memoria è separata rispetto alla gestione delle query. A destra, la struttura dei tradizionali DSMS. 73
3.2 Parametro 'Role'. 75
3.3 Produttore e Consumatore. 79
3.4 Esempi di query per ciascuna modalità di accesso. 80
3.5 Esempi di query per ciascuna modalità di accesso. 82
3.6 Esempio di store condiviso. 84
3.7 Struttura dati che implementa uno store RANDOM. 86
3.8 Struttura dati che implementa uno store FIFO. 88
3.9 Costruzione dell'indice sui dati. 90
3.10 Sintassi astratta per L. Descrizione dei simboli e dei domini. . . . 94
3.11 Sampling method. 103

4.1 Simulazione dello stream proveniente dalle OBU. 113
4.2 Processo di esecuzione di una query. 115
4.3 Architettura del prototipo DSMS sviluppato. 116
4.4 Tre diversi consumatori che agiscono su tre finestre temporali diverse. 119
4.5 Use Case 1. Esecuzione della query produttore da parte di un utente. 122
4.6 Use Case 2. Sottomissione di un query consumatore da parte di un utente. 124
4.7 Grafo delle classi. 127
4.8 GUI utente. 134

5.1 Andamento dei tempi di risposta massimi al variare del numero di thread e del dataset. Si vede che, aumentando la quantità di informazioni che il produttore legge da file e il numero di thread, i tempi di risposta aumentano. Il polling time del produttore è uguale a 1 secondo. 144

Introduzione

Oggigiorno l'uso di tecnologie dell'informazione a supporto della mobilità e degli spostamenti di persone e merci è sempre piu richiesto.
La possibilità di inviare, raccogliere ed elaborare dati ed informazioni a supporto di chi si muove (in auto, a piedi, o in qualsiasi altro modo) è attuabile per mezzo di una serie di strumenti e tecniche in rapida e continua diffusione. Conoscere in anticipo la situazione della viabilità permette, ad esempio, una migliore pianificazione dei propri spostamenti, riducendo sensibilmente il problema delle code e dei conseguenti disagi (stress, inquinamento, ecc.). Numerose città usano le tecnologie di infomobilità a questo scopo.
La parola Infomobilità, secondo la definizione del gruppo *Intelligent Energy* della Commissione Europea, indica tutta quella serie di procedure, sistemi e strumenti basati sull'*ITS (Intelligent Transport System)* che permettono di migliorare la mobilità di persone e merci grazie alla raccolta, elaborazione e distribuzione di informazioni.
I progressi dei servizi di infomobilità seguono gli sviluppi della tecnologia dell'informazione. Le applicazioni possono essere utilizzate sia dagli operatori della mobilità sia dagli utenti finali dei servizi di mobilità.
I servizi di infomobilità costituiscono un supporto per la gestione dinamica del trasporto pubblico, del traffico stradale, delle flotte di veicoli e delle infrastrutture di trasporto grazie alla disponibilità di informazioni in tempo reale. Grazie ai sistemi ITS sono stati sviluppati sistemi informativi integrati dedicati al trasporto dove tutti gli elementi sono reciprocamente interconnessi allo scopo di avere più efficienza, più sicurezza e in qualche misura più sostenibilità ambientale.
Tra i soggetti che possono usufruire dei servizi forniti dall'infomobilità sono presenti:

- *Viaggiatori*: I servizi offerti permettono al viaggiatore non solo di pianificare il percorso prima di partire da casa ma anche durante il viaggio in maniera dinamica. In particolare si può proporre al viaggiatore percorsi alternativi in caso di traffico o incidenti nelle vicinanze, offrirgli informazioni su ristoranti o locali nelle vicinanze, aggiornarlo su eventi di interesse e tanto altro.

- *Enti locali*: Gli enti locali possono sfruttare i dati provenienti dal territorio per fornire informazioni di vario tipo ai cittadini e razionalizzare gli spostamenti su larga scala.

- *Aziende di trasporto e logistica*: Fondamentale per un azienda di trasporto è tenere sotto controllo gli spostamenti e la posizione di mezzi e merci in modo da migliorare l'efficienza di trasporto e la trasparenza nei confronti del cliente.

In questo contesto si inserisce il progetto *PEGASUS* (ProgEtto per la Gestione della mobilità Attraverso Sistemi infotelematici per l'ambito Urbano, per la Sicurezza di passeggeri, veicoli e merci) il cuo obbiettivo è la realizzazione di una piattaforma infotelematica integrata per migliorare la mobilità cittadina. Le aree urbane del Paese sono infatti sempre più congestionate dal traffico, con conseguente aumento degli incidenti, dei rallentamenti e dell'inquinamento atmosferico/acustico. La piattaforma dovrà fornire, in particolare, un servizio di navigazione intelligente basato sulla raccolta ed elaborazione in tempo reale dei dati provenienti da apposite *OBU (On Board Unit)* che, installate sui veicoli, trasmetteranno informazioni sulla loro velocità per consentire la rilevazione di eventuali rallentamenti o ingorghi. Per la piattaforma dovrà essere sviluppato un *middleware* database in grado di ricevere, memorizzare e gestire le informazioni sul traffico in tempo reale e dotato di un innovativo algoritmo di ricerca e inserimento di eventi (incidenti, code) che alterano la normale viabilità.
L'obbiettivo della tesi è stato quello di studiare e sviluppare un sistema innovativo per la gestione efficiente dei dati provenienti dalle *OBU* che permetta di separare gli aspetti di gestione e indicizzazione da quelli di interrogazione del dato.
I moderni *DSMS (Data Stream Management System)* tendono ad accoppiare il sistema di *Storage Management*, che si occupa della memorizzazione del dato, con quello di *Query Processing*, che si occupa dell'interrogazione del dato, e di

conseguenza il sistema diventa rigido e scarsamente ottimizzabile. La soluzione migliore risiede nel disaccoppiare i due sistemi in modo da permettere allo Store Manager di gestire al meglio i dati e fornire al Query Processing le primitive per poter accedere rapidamente a tali dati. Inoltre il sistema risulta maggiormente flessibile e ottimizzabile sulla base delle query che si vogliono eseguire.

La presente Tesi è strutturalmente organizzata in cinque capitoli, suddivisi in due parti, la **Prima** delle quali riporta gli argomenti studiati per sviluppare il lavoro di ricerca e implementazione presentato nella **Seconda Parte**.

Nel **Capitolo 1** viene descritto il progetto nel quale si inserisce la Tesi. Il **Capitolo 2** servirà a descrivere gli aspetti principali di un DSMS, gli scenari di applicazione tipici e i principali DSMS presenti oggigiorno sul mercato. Il **Capitolo 3** inizia la seconda parte di trattazione e descrive in dettaglio l'architettura di un innovativo DSMS in cui la parte di Store Management e quella di Query Processing Engine sono separate. Il **Capitolo 4** descrive la struttura e il funzionamento del prototipo software sviluppato sulla base dei requisiti funzionali richiesti. Il **Capitolo 5** presenta infine le prove sperimentali eseguite sul sistema realizzato e i risultati ottenuti.

Parte I

Stato dell'arte

Capitolo 1

Progetto PEGASUS

L'obbiettivo di quanto segue in questo capitolo è descrivere e dettagliare gli obbiettivi del progetto PEGASUS e l'architettura di funzionamento.

1.1 Motivazioni e obiettivi del progetto

L'elevata congestione del traffico ha ormai convinto sia gli enti di governo sia l'opinione pubblica che il miglioramento della qualità della vita nelle città si può ottenere solamente con un'ottimizzazione della mobilità urbana.
I problemi ambientali causati dall'incremento del traffico urbano e dal sistema attuale infrastrutturale dei trasporti, sia in Europa che in Italia, sono principalmente legati alla difficoltà di soddisfare la crescente domanda di mobilità con le infrastrutture esistenti. In questo senso le ricadute più evidenti sono sulla qualità della vita delle persone, sull'ambiente urbano e sulla congestione veicolare stradale. In prima battuta, la soluzione a questo tipo di problemi viene ricercata intervenendo sulla dotazione di infrastrutture, sia aumentandone il numero che ampliandone la capienza. Ma, com'è noto, tale approccio, che porterebbe ad adeguare qualsiasi struttura ai suoi massimi picchi di richiesta, non è perseguibile se non affrontando enormi costi economici e, nel caso delle infrastrutture viarie e di trasporto, con particolari impatti negativi anche sul territorio per una pluralità di aspetti (paesaggio, rottura di equilibri geofisici, ecc.). Il problema della mobilità urbana deve essere affrontato con urgenza in quanto si valuta che gli ingorghi cittadini in Europa producano costi esterni pari allo 0.5% del Prodotto Interno Lordo.

La soluzione migliore risiede nell'investire sulle tecnologie e sui sistemi per l'infomobilità in modo da ottimizzare e supportare gli spostamenti di persone o merci al fine di migliorare la qualità della vita delle persone, diminuire i consumi e salvaguardare l'ambiente.
Il supporto ad una mobilità sostenibile risulta essere una delle maggiori sfide nell'ambito di ricerca dell'ICT e racchiude diverse aree di ricerca come il *GIS (Geographical Information System)* [23], il *Networking* [8], l'*Operation Research* [18], le *Wireless Sensor Network* [27] e le *Telecomunicazioni* [17].
In questo contesto si inserisce il progetto PEGASUS (ProgEtto per la Gestione della mobilità Attraverso Sistemi infotelematici per l'ambito Urbano, per la Sicurezza di passeggeri, veicoli e merci) il cuo obbiettivo è la realizzazione di una piattaforma infotelematica integrata per migliorare la mobilità cittadina.
L'obbiettivo finale sarà creare un sistema intelligente di monitoraggio e controllo del traffico urbano in modo da rilevare eventuali code o incidenti e proporre in real-time percorsi alternativi ai veicoli, gestire dinamicamente le mappe includendo i POIs (Points Of Interest) (ristoranti, locali, ecc.) segnalati dai viaggiatori ed i EOI (Events Of Interests) (code, rallentamenti, ecc.) garantendo un miglioramento della mobilità.

1.2 Descrizione generale del progetto

Il progetto PEGASUS prevede l'installazione su ciascun veicolo di un dispositivo denominato OBU (On Board Unit) in grado di rilevare, per mezzo di alcuni sensori, la velocità, l'accellerazione e, tramite un ricevitore GPS, la sua posizione. Queste informazioni verranno comunicate in real-time ad intervalli regolari al centro di controllo che le utilizzerà per rilevare eventuali anomalie e proporre delle soluzioni.

Le OBU non comunicano solo con il centro di controllo ma anche tra di loro [11, 12, 16]. Vengono creati dinamicamente dei cluster [11] in cui il master raccoglie e aggrega i dati relativi a tutti i veicoli nel cluster e li invia al centro di controllo. Questa strategia viene utilizzata per minimizzare le comunicazioni V2I (Vehicle to Infrastructure) che risultano molto costose in quanto sfruttano la tecnologia a pagamento GPRS e favorire le comunicazioni V2V (Vehicle to Vehicle) in quanto sfruttano la tecnologia di comunicazione gratuita Wi-Fi.
Uno degli aspetti più complessi del progetto risulta la gestione efficiente degli

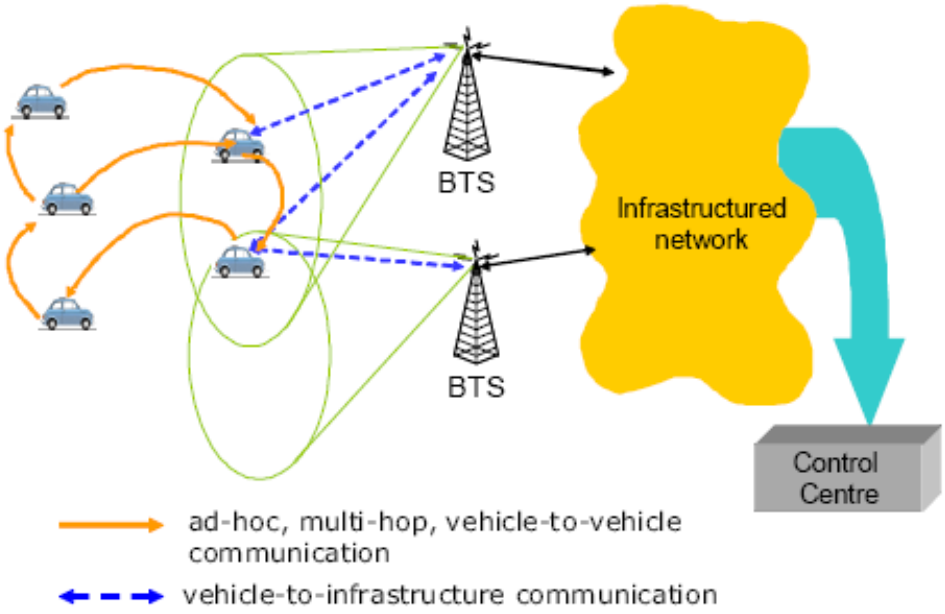


Figura 1.1: Infrastruttura progetto PEGASUS

stream di dati in arrivo dalle OBU. Il centro di controllo deve essere in grado di memorizzare ed interrogare in real-time una grosse mole di dati in cui la scalabilità e i bassi tempi di risposta sono un requisito fondamentale. A questo scopo i dati dovranno essere gestiti in maniera opportuna, ad esempio separando i dati 'recenti', che permettono di rilevare in maniera tempestiva gli incidenti e gli ingorghi, da quelli 'storici' che invece sono utili per fare stime sui tempi di percorrenza, flussi medi, ecc. In questo modo si cerca di separare le informazioni che vanno gestite immediatamente da quelle che possono essere elaborate in un secondo momento. Le informazioni recenti hanno requisiti *hard real-time* e servono per avvisare tempestivamente gli automobilisti di eventuali anomalie e offrirgli velocemente soluzioni alternative. I dati storici invece possono essere utilizzati per effettuare statistiche, profiling degli utenti ed effettuare predizioni sul traffico futuro. Per questo motivo, all'interno del centro di controllo, è stata progettata una architettura DSMS in grado di gestire in maniera opportuna ed efficiente lo stream di dati in arrivo. Il DSMS deve essere inoltre in grado di gestire i POI segnalati dagli utenti e il cui scopo finale è creare un *Recommender System* collaborativo [2] in modo da visualizzare sulla mappa dei viaggiatori i vari POI che potrebbero interessargli con le valutazioni fornite dagli utenti stessi.

1.2.1 OBU

Nel sistema analizzato, ciascun veicolo sarà equipaggiato con un dispositivo OBU. Questo dispostivo è in grado di elaborare e comunicare diverse informazioni ed è composto da diverse unità il cui scopo è raccogliere differenti informazioni sul veicolo, tra cui la velocità, l'accellerazione e la posizione.

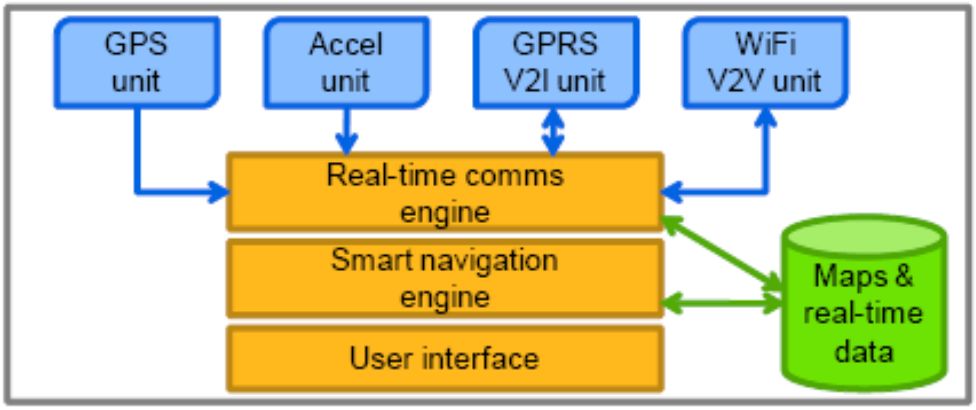


Figura 1.2: Architettura On-Board Unit

In particolare il dispositivo è composto da:

- *Unità GPS*: Sistema di posizionamento su base satellitare. Serve per rilevare la posizione del veicolo.
- *Accellerometro*: Strumento in grado di rilevare l'accellerazione del veicolo.
- *Unità di comunicazione V2I GPRS*: Unità in grado di comunicare con la Centrale Di Controllo utilizzando la tecnologia di telefonia mobile cellulare GPRS. La massima velocità di una connessione GPRS è circa uguale a quella ottenibile tramite modem collegato alla normale rete telefonica analogica, ovvero fra 4 e 5 kB/s, a seconda del modem impiegato. Il tempo di latenza è piuttosto alto: per averne un ordine di grandezza si consideri che un'operazione di ping (acronimo di *Packet INternet Groper*) dura di solito 600-700 ms (a volte anche 1 s). Le tariffe applicate dai vari gestori variano enormemente (da 1 a 20 euro per megabyte). Negli USA la T-Mobile offre un abbonamento a 30$ al mese senza limitazioni di tempo. Anche la AT&T Wireless offre abbonamenti di tipo flat. Questi parametri

devono essere considerati in fase di progettazione del sistema in quanto le comunicazioni GPRS devono essere minimizzate risparmiando su costi e intasamento di rete.

- *Unità di comunicazione V2V Wi-Fi*: Unità che permette di comunicare con gli altri veicoli per mezzo della tecnologia gratuita wireless.
- *Real-Time Communication Engine*: Strato logico che si occupa della gestione di tutte le comunicazioni in input/output dall'OBU. Implementa le politiche di comunicazione V2I e V2V presenti nel dispositivo. Dovrà interagire: con l'unità di comunicazione V2V Wi-Fi per ricevere le informazioni sugli altri veicoli; con l'unità di comunicazione V2I GPRS per ricevere informazioni dal centro di controllo; con l'unità GPS per informazioni sulla posizione attuale; con l'accellerometro; con la memoria flash presente sul dispositivo che contiene le mappe statiche e i dati dinamici ricevuti dalle altre OBU e dal centro di controllo, i quali sono gestiti e modificati direttamente dal modulo stesso.
- *Smart Navigation Engine*: Accede alla memoria flash e, intersecando le informazioni sulle mappe statiche con i dati dinamici, fornisce i servizi ai viaggiatori.
- *User Interface*: Interfaccia grafica per l'interazione con l'utente.
- *Memoria Flash*: Memoria che contiene: i dati statici relativi alla topologia delle mappe; i dati dinamici relativi alle altre OBU all'interno del cluster; i dati forniti dal centro di controllo tramite GPRS.

I dati che verranno acquisiti e gestiti dalle OBU sono:

- Posizione, Velocità e Tempi di percorrenza. Queste informazioni sono fondamentali per la gestione rea-time del traffico.
- Accellerazioni/Decellerazioni fornite dall'accellerometro. Queste informazioni sono utili per rilevare incidenti o intasamenti.
- Notifiche di POI/EOI e le valutazioni sui POI fornite dagli utenti.

Le OBU informano in real-time il centro di controllo sulle notifiche relative ai POI/EOI e con i dati ricevuti dal GPS.

Dato che i costi di comunicazione GPRS sono abbastanza alti vengono seguite delle politiche innovative di *communication-saving* in cui si cerca di minimizzare le comunicazioni V2I. Spesso le politiche di comunicazione V2V e V2I vengono combinate e selezionate in maniera dinamica sulla base delle condizioni e dei requisiti del centro di controllo in modo da scegliere la soluzione che minimizza i costi globali di comunicazione e massimizza l'efficacia dei servizi offerti.

1.2.2 Centro di controllo

Nel progetto PEGASUS il centro di controllo si occupa di gestire ed utilizzare gli stream di dati in arrivo dai veicoli allo scopo di fornire i servizi per la mobilità. I servizi offerti possono riguardare la *Smart Navigation* (navigazione intelligente), la *Urban Mobility* (mobilità urbana) e la *Safety*(sicurezza).
Il centro di controllo comunica con i veicoli tramite GPRS e riceve informazioni di vari tipo tra cui la velocità, posizione e segnalazione di eventuali POI da parte degli automobilisti. I POI/EOI sono gestiti mediante ontologie, in modo da offrire un livello di astrazione superiore rispetto ai dati 'grezzi' in arrivo dalle *OBU*. Queste ontologie possono essere ampliate dinamicamente sulla base dei dati che arrivano dai veicoli e permettono di raccogliere tutti i punti di interesse segnalati dagli utenti in modo da poterli proporre ad altri utenti sulla base delle loro preferenze. Al centro di controllo arriverà una grande quantità di dati che dovrà essere gestita tenendo conto dei requisiti di performance richiesti dai servizi. In particolare, le operazioni eseguite sui dati recenti avranno requisiti prestazionali differenti rispetto alle operazioni eseguite sui dati storici in quanto i dati recenti verranno utilizzati per rilevare tempestivamente situazioni anomale e prendere velocemente delle decisioni, mentre i dati storici verranno utilizzati per effettuare analisi statistiche e predizioni future, pertanto non ci sono vincoli stringenti di performance da dover rispettare. I dati recenti diventano storici dopo un certo intervallo di tempo che può variare a seconda dell'applicazione e delle query che vengono eseguite.
Nel sistema ci saranno delle query continue che, a differenza delle query classiche *one-shot* dei DBMS, vengono eseguite ad intervalli regolari sullo stream di dati in input e servono per monitorare costantemente il traffico. Maggiori dettagli sul centro di controllo verranno forniti nella prossima sezione.

1.2.3 Comunicazioni V2V

Nel sistema analizzato le OBU, oltre che comunicare con il centro di controllo, comunicano anche tra di loro.
Questo avviene essenzialmente per due motivi:

- **Diminuzione dei costi GPRS**: Le comunicazioni verso il centro di controllo sfruttano la tecnologia GPRS che, essendo a pagamento e non particolarmente veloce, deve essere ottimizzata. Ovvero si cerca di minimizzare il numero di comunicazioni totali e massimizzare l'efficienza dei servizi offerti. Le comunicazioni V2V mirano ad aggregare le informazioni relative a più veicoli ed effettuare una sola comunicazione verso il centro di controllo.

- **Miglioramento delle prestazioni**: Le comunicazioni V2V sfruttano la tecnologia di comunicazione gratuita Wi-Fi che può essere sfruttata per aggiornare le OBU sulla situazione del traffico nella vicinanze senza bisogno di comunicare con il centro di controllo, diminuendo i costi e migliorando notevolmente le performance.

Le tecniche di comunicazione V2V cercano di creare in maniera dinamica dei cluster auto-organizzati di OBU e aggregare le informazioni all'interno del cluster in modo da minimizzare le comunicazioni V2I. Le caratteristiche principali dei cluster sono una forte dinamicità, mobilità e presenza di errori (collisioni o pacchetti persi/danneggiati). Inoltre è possibile che si verifichino delle *Broadcast Storm* (tempesta di trasmissioni), ovvero situazioni in cui nella rete, per ogni messaggio che viene inviato, si richiede al ricevente di rispondere inoltrando a sua volta un messaggio. Questo fenomeno causa un aumento esponenziale del traffico di rete che, quando non la satura completamente, ne riduce drasticamente le performance.
Le tecniche di comunicazione V2V prevedono che all'interno del cluster ci siano dei *Cluster Member (CM)* che inviano le informazioni ad un *Cluster Head (CH)* il quale si occupa di aggregare le informazioni e inviarle al centro di controllo.
Le comunicazioni intra-cluster sono molto veloci e permettono alle OBU di reagire velocemente a eventuali situazioni anomale che si verificano all'interno del cluster. Infatti le OBU del cluster possono essere informate immediatamente di un EOI tramite un messaggio di broadcast inviato dal CH che si occuperà anche di informare il centro di controllo. Inoltre, per quanto riguarda il monitoraggio

del traffico, eseguire all'interno del cluster un protocollo di *dynamic distributed aggregation* consente di eseguire funzioni che misurano alcune informazioni utili come il numero medio di veicoli (per mezzo del *dynamic counting*) o la velocità media delle vetture (per mezzo del *distributed averaging*) [16].

1.2.4 Comunicazioni V2I

Le comunicazioni verso il centro di controllo, come già espresso precedentemente, sfruttano la tecnologia di telefonia mobile cellulare GPRS. Per questo motivo il numero totale di comunicazioni deve essere minimizzato. I dati che generalmente vengono inviati/ricevuti dal centro di controllo sono relativi alle segnalazione di POI/EOI e all'aggiornamento su posizione, velocià e altri parametri delle OBU. Le tecniche di *communication-saving V2I* prevedono che i dati relativi ai POI/EOI possano essere semplicemente inviati sulla base delle richieste degli utenti mentre i dati relativi alla posizione, velocità e altre informazioni fornite dal GPS devono essere inviati in automatico con una certa frequenza in modo da avere un quadro completo e aggiornato della situazione e garantire l'efficienza dei servizi offerti.
Le politiche per l'invio automatico dei dati possono essere:

- *Sampling Based*: I dati vengono inviati con una certa frequenza temporale o/e spaziale, ad esempio ogni minuto o ogni 2km. Questa frequenza può essere ad intervalli regolari (**Simple Sampling**) o irregolari, ovvero basati sulla posizione dei veicolo sulla mappa (**Map-Based Sampling**), ad esempio aumentando la frequenza di invio nelle zone di maggior traffico (centri urbani, strade principali, ecc.) e diminuendo nelle zone poco congestionate (strade extraurbane, zone periferiche, ecc). In genere questa politica permette di avere un quadro molto preciso sulla posizione dei veicoli ma le comunicazioni totali possono diventare notevoli.

- *Information Based*: Questa politica cerca di minimizzare le comunicazioni tenendo traccia in maniera approssimata della posizione dei veicoli sulle varie strade. Nel caso si adotti una politica **Deterministic Information-need** il centro di controllo invia in broadcast alle OBU la velocità v_b prevista per ciascuna strada di interesse e le OBU trasmettono la propria velocità v_m solo se $|v_b - v_m| > T$ dove T è una certa soglia specificata [12].

Nel caso invece si adotti una politica **Probabilistic Information-need** ciascuna OBU invia le proprie informazioni con una certa probabilità p. Queste politiche possono essere adottate entrambe e si può usare l'una o l'altra in maniera dinamica a seconda della situazione presente.
Infine, per le interazioni V2I, si possono adottare tecniche già previste per le wireless sensor network, come il *packet merging* [25] (inviare un pacchetto grande è piu economico che inviare tanti pacchetti piccoli) e la *linear regression* [14] (sfrutta le letture ridondanti dai veicoli per diminuire le comunicazioni).

1.3 Centro di controllo

1.3.1 Descrizione architettura

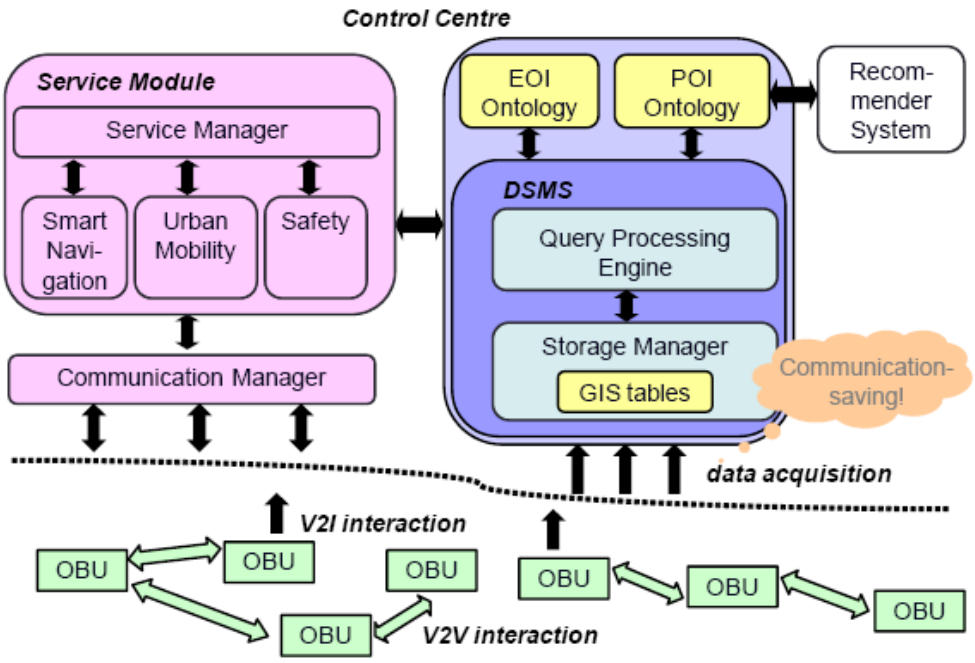


Figura 1.3: Architettura centro di controllo del progetto PEGASUS

L'architettura del centro di controllo è suddivisa in due grossi moduli: il *Data Stream Management System* e il *Service Module*, come mostrato in figura 1.3.

Il Service Module interagisce con il DSMS per ricevere le informazioni che gli serviranno per rispondere alle richieste di servizio degli utenti. Inoltre è presente un modulo denominato *Communication Manager* che gestisce le comunicazioni tra le OBU e il Service Module.

1.3.2 Service Module

Il Service Module si occupa di gestire i servizi che verranno offerti agli utenti. E' composto da diversi moduli, tra cui:

- *Service Manager*: Modulo che interagisce con tutti gli altri moduli del Service Module per gestire al meglio i servizi offerti.
- *Smart Navigation*: Offre funzionalità per i servizi che richiedono la navigazione intelligente. La navigazione intelligente potrà essere garantita attraverso l'invio di informazioni sul traffico in tempo reale ai veicoli, consentendo così al navigatore di bordo di aggiornare continuamente il percorso ottimo.
- *Urban Mobility*: Offre servizi per la mobilità urbana. L'obbiettivo di questo modulo è migliorare il traffico urbano, razionalizzando gli spostamenti e gestendo al meglio i flussi. L'Urban Mobility permetterà di avere un sistema di gestione di nodi e servizi strategici della città, come i permessi di transito/circolazione, la disponibilità dei parcheggi in tempo reale e la gestione del flusso delle merci all'interno della rete stradale cittadina.
- *Safety*: Modulo che offre servizi che migliorano la sicurezza del viaggiatore, come ad esempio la tempestiva segnalazione di un incidente e la conseguente immediata risposta dei servizi di emergenza.

1.3.3 Data Stream Management System (DSMS)

Una delle parti più importanti del centro di controllo risiede nel DSMS che è responsabile dell'acquisizione, memorizzazione e manipolazione degli stream di dati in arrivo dalle OBU. Questi dati saranno geo e tempo localizzati. La scalabilità e le performance richieste dal sistema possono essere garantite solo se i dati

vengono gestiti in maniera efficiente. Inoltre devono essere previste le ottimizzazini delle query, il cui scopo e' quello di produrre un 'piano di esecuzione' il piu' possibile efficiente, basandosi su quanto e' specificato dall'albero della query. I dati, all'interno del DSMS, dovranno essere gestiti in maniera flessibile e differenziata a seconda che si tratti di dati recenti o dati storici. I dati recenti dovranno essere gestiti in memoria principale e indicizzati in opportune strutture dati in modo che l'accesso ad essi sia rapido, mentre i dati storici, che generalmente non hanno un utilità immediata, potranno essere migrati su memoria secondaria, ad esempio all'interno di un DBMS, e serviranno per effettuare analisi statistiche e predizioni sul traffico futuro in modo da gestire preventivamente il traffico.

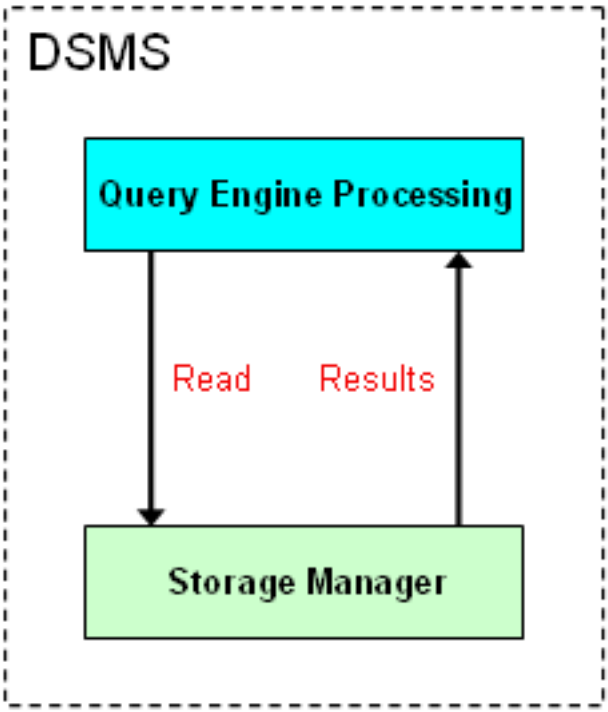


Figura 1.4: Interazione tra Store Manager e Query Processing Engine

Come mostrato in figura 1.4, il DSMS è un sistema composto da due moduli, il Query Processing Engine (QPE), che si occupa della ricezione e ottimizzazione delle query da eseguire e lo Storage Manager (SM), che si occupa della gestione e indicizzazione dei dati in arrivo dai veicoli e dei dati storici [1, 4, 9]. L'SM

dovrà inoltre offrire al QPE una opportuna interfaccia di comunicazione per poter recuperare i dati.

Query Processing Engine

Il Query Processing Engine (QPE) è quella parte del DSMS che si occupa dell'ottimizzazione e gestione delle query da eseguire nel sistema. Sugli stream di dati in ingresso dovranno essere eseguire delle query continue per monitorare lo stato del traffico e rilevare eventuali situazioni anomale.
Le query che vengono sottomesse al QPE vengono ottimizzate, parserizzate e trasformate in una serie di comandi di lettura presso lo Store Manager, il quale restituirà i dati richiesti. Queste query vengono anche chiamate query 'consumatore', in quanto 'consumano' le tuple presenti nello Store Manager che potranno quindi essere successivamente gestite come dati storici in quanto non più utili per le operazioni real-time. Le query possono interrogare sia i dati recenti che quelli storici.
La caratteristica principale del Query Processing Engine è il fatto di dover gestire query molto differenti rispetto alle normali query di un DBMS tradizionale. La prima grossa differenza riguarda la tipologia di dati che bisogna interrogare. Bisognerà interrogare dati referenziati spazialmente e temporalmente. Questo necessita di query in cui la clausola WHERE deve essere applicata su una certa *finestra temporale*, superata la quale il dato è considerato troppo 'vecchio' e quindi non più valido. Potranno essere sottomesse query del tipo: forniscimi il numero totale di vetture che hanno attraversato la strada X nell'ultima ora; dato un punto, calcolami il numero di veicoli che si trovano all'interno di un certo raggio da quel punto; ecc.. Inoltre bisognerà poter definire query continue che si ripetono ad intervalli regolari in modo da monitorare costantemente il traffico di dati in input.
Infine la possibilità di eseguire più query contemporaneamente, fornire risposte approssimate e ottimizzare il query plan dinamicamente rientra tra le caratteristiche principali di un motore QPE in cui le performance e la scalabilità sono gli obbiettivi principali.

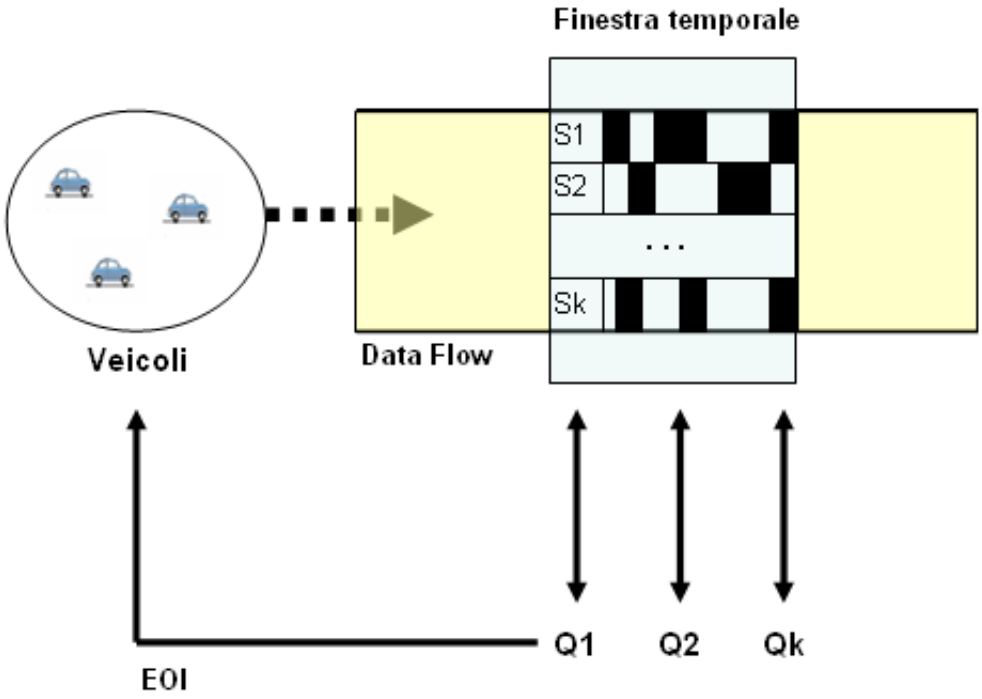


Figura 1.5: Il data stream in arrivo dalle OBU viene inserito una finestra temporale su cui verranno eseguite le query.

Storage Manager

Lo Storage Manager è quella parte del DSMS che si occupa della gestione dello stream di dati in arrivo dalle OBU. I dati sono georefenziati e temporalmente ordinati, pertanto l'utilizzo di un GIS (Geographic(al) Information System) risulta fondamentale. Il GIS consente di mettere in relazione tra loro dati diversi, sulla base del loro comune riferimento geografico in modo da creare nuove informazioni a partire dai dati esistenti. I GIS presentano normalmente delle funzionalità di analisi spaziale ovvero di trasformazione ed elaborazione degli elementi geografici degli attributi.
Esempi di queste elaborazioni sono:

- *Overlay topologico*: Si effettua una sovrapposizione tra gli elementi dei due temi per creare un nuovo tematismo (ad esempio per sovrapporre il tema dei confini di un parco con i confini dei comuni per determinare le superfici

di competenza di ogni amministrazione o la percentuale di area comunale protetta).

- *Query spaziali*: Sono interrogazioni alle basi di dati che sfruttano criteri spaziali (vicinanza, inclusione, sovrapposizione, intersezione, ecc.). Questa tipologia di query risulta di fondamentale importanza nel sistema in esame in quanto, ad esempio, rilevare le vetture vicine al luogo di un incidente per inviargli una notifica, valutare le distanze da un punto di interesse e calcolare i tempi di percorrenza sono tra i requisiti del progetto.

- *Buffering*: E' una regione di una specifica grandezza costruita intorno ad un punto, una linea o un area. Ad esempio una query effettuata in PostGIS del tipo:

  ```
  SELECT BUFFER((SELECT thegeom
  FROM tubature
  WHERE nome='pincopallino'), 2);
  ```

  Restituisce il luogo dei punti con distanza inferiore o uguale a 2 dalla geometria selezionata.
  In genere queste elaborazioni vengono utilizzate per le analisi di prossimità.

- *Segmentazione dinamica*: Sono algoritmi che di solito sono applicati su temi lineari per determinare un punto ad una determinata lunghezza dall'inizio del tema. Gli elementi lineari, in una visione del mondo di tipo geografico, raffigurano oggetti come strade, fiumi o limiti amministrativi. Utilizzando l'estensione al modello dati di tipo georelazionale chiamata segmentazione dinamica, è possibile rappresentare tali elementi lineari ed associare delle informazioni (gli attributi) a qualsiasi porzione degli archi che rappresentano tali elementi. La rappresentazione all'interno di un GIS di elementi geografici come un reticolo stradale o fluviale viene effettuata memorizzando una serie di coordinate X, Y in un dato sistema di riferimento. Un altro metodo per identificare la posizione di un elemento geografico potrebbe essere anche la sua posizione lungo una strada od un fiume (ad esempio: al 12esimo chilometro dell'autostrada A14). Questo metodo semplifica enormemente l'acquisizione dei dati, in quanto consente di memorizzare e gestire solamente una coordinata di posizione invece che due e di usare un sistema di riferimento più vicino alla realtà dell'utilizza-

tore finale.
Le funzionalità di segmentazione dinamica permettono quindi di rappresentare e gestire in maniera estremamente efficace delle informazioni associabili ad elementi geografici lineari.

- *Network Analysis*: Sono algoritmi che da una rete di elementi lineari (es. rete stradale) determinano i percorsi minimi tra due punti. Anche questa funzionalità risulta fondamentale nel progetto in quanto, ad esempio, per i servizi che riguardano la smart navigation calcolare il percorso minimo tra un punto di partenza e uno di arrivo è un requisito richiesto.

- *Spatial Analysis*: Sono algoritmi che permettono di creare, interrogare ed analizzare dati raster ed eseguire analisi integrate tra dati raster e dati vettoriali. Le analisi della distanza tengono conto dell'orografia (branca della geografia fisica che studia i rilievi della Terra), della presenza di zone più agevolmente transitabili, di ostacoli naturali e artificiali per individuare i percorsi più brevi e accessibili tra determinati punti.

Tutti questi aspetti rendono l'utilizzo di un GIS fondamentale.
Un'altra caratteristica dei dati di cui lo Storage Manager deve tenere conto è la loro alta variabilità temporale. In pratica non si devono gestire dati statici come generalmente avviene in un DBMS tradizionale ma dati che variano continuamente nel tempo e la cui 'importanza' cambia nel tempo. Questa caratteristica rende impossibile la gestione di tali dati tutti in memoria primaria perchè gli stream di dati in ingresso sono potenzialmente infiniti, pertanto serviranno politiche che permettano di differenziare i dati e gestirli in maniera diversa. Ad esempio, bisognerà differenziare i dati recenti da quelli passati, in quanto sui dati recenti generalmente si hanno requisiti hard real-time (rilevazioni di incidenti, situazioni anomale, segnalazione di emergenze, ecc.), mentre i dati storici serviranno per analisi statistiche e predizioni future [5].

POI ed EOI

Il DSMS fa uso di ontologie per la gestione dei POI e degli EOI. Un'ontologia è una rappresentazione formale, condivisa ed esplicita di una concettualizzazione di un dominio di interesse. Permette di descrivere le parole comuni e i concetti (significati) usati per descrivere e rappresentare un'area di conoscenza (dominio).

La parte di gestione delle ontologie è separata rispetto al DSMS in modo da rendere il DSMS indipendente dal contesto applicativo nel quale lo si inserisce. Le ontologie permettono una gestione flessibile ed estendibile dei punti di interesse e degli eventi e aggiunge un livello di astrazione rispetto ai dati 'grezzi' in arrivo dalle OBU, in modo da categorizzare nello stesso modo informazioni che sono molto simili tra di loro. Il dominio ontologico relativo ai POI ed EOI può crescere dinamicamente in quanto nuovi concetti possono venire aggiunti e popolati con i nuovi dati in arrivo. I dati relativi ai POI possono essere gestiti come dati storici in quanto, analizzando la storia di una particolare OBU, si possono descrivere e definire i gusti e comportamenti tipici degli utenti.
Per quanto riguarda gli EOI vengono modellati molti eventi di interesse (incidenti, code, lavori, ecc.) che servono a definire le condizioni del traffico. Gli eventi sono individuati analizzando le informazioni ricevute dalle OBU, ad esempio analizzando la velocità e il trend di accellerazioni si può capire se è presente un coda oppure no; se due veicoli vicini hanno velocità nulla per un lungo periodo di tempo potrebbe essersi verificato un incidente. Inolte gli eventi possono essere rilevati utilizzando dei metodi simulativi o predittivi, come le *DPM (Dynamic Probabilistic Models)* o le *HMM (Hidden Markov Models)* [21]. Quest'ultimo metodo, in particolare, data una sequenza di osservazioni sulla velocità del traffico presente, stimano lo stato di traffico futuro più probabile. La bontà delle predizioni viene calcolata osservando le distanze tra il valore predetto e quello reale.
Le ontologie relative agli eventi descrivono l'evento stesso e le regole che generano l'evento (*trigger condition*). L'evento 'scatta' quando le informazioni raccolte dalle OBU rispettano le regole definite dall'ontologia relativa a quell'evento. La rilevazione di un evento fa eseguire all'interno del DSMS le query legate a quell'evento, come ad esempio il monitoraggio della zona in cui si è verificato l'incidente e la comunica ad un operatore, e continuano ad eseguire fin tanto che l'evento non si è risolto (*stop condition*). Quindi le ontologie sugli eventi avranno anche delle condizioni che definiscono la risoluzione dell'evento stesso e di conseguenza di tutte le query ad esso associate. Ad alto livello si potrebbe rappresentare questo concetto in questo modo:

```
QUERY: < Monitoraggio del traffico in un raggio di 500m
dall'incidente >
```

```
TRIGGER: < Vetture ferme nello stesso punto per più di 5 minuti >
STOP CONDITION: < Le vetture tornano in movimento >
```

Questa query può essere inserita, ad esempio, nell'ontologia 'Incidente'.

1.3.4 Gestore delle comunicazioni

Il Communication Manager si interpone tra gli utenti e il Service Module. Il suo obbiettivo è fornire un interfaccia standard e trasparente per l'accesso ai servizi e si occupa di organizzare ed instradare le richieste verso il servizio più opportuno. Questo strato aggiuntivo tra gli utenti e il Service Module permette di aumentare l'astrazione dei servizi offerti e gestire il Service Module come una *Service-Oriented Architecture (SOA)*. Una Service-Oriented Architecture e' un modello architetturale per la creazione di sistemi residenti su una rete che focalizza l'attenzione sul concetto di servizio (come ad esempio la Smart-Navigation, i servizi legati alla Safety, l'Urban-Mobility, ecc.). Un sistema costruito seguendo la filosofia SOA è costituito da applicazioni, chiamate servizi, ben definite ed indipendenti l'una dall'altra, che possono risiedere su più computer all'interno di una rete. Ogni servizio mette a disposizione una certa funzionalità e può utilizzare quelle che gli altri servizi hanno reso disponibili, realizzando, in questo modo, applicazioni di maggiore complessità. L'astrazione delle SOA permette notevoli vantaggi, come ad esempio la possibilità di definire interfaccie indipendenti dall'implementazione specifica. Deve cioè essere definito in termini di ciò che fa, astraendo dai metodi e dalle tecnologie utilizzate per implementarlo. Inoltre aggiungere servizi può essere fatto semplicemente pubblicando la propria interfaccia in un Service Registry o Service Directory, rendendo l'accesso ad esso trasparente rispetto alla locazione.
Tutti questi aspetti permettono quindi all'utente di riciedere un servizio in maniera semplice e al centro di controllo di aggiungere nuovi servizi più facilmente (basterà pubblicare la nuova interfaccia).

1.3.5 Recommender System

Nello Storage Manager si tiene traccia di tutti i POI segnalati dagli utenti e il relativo rating. Queste informazioni, combinate con il profilo degli utenti [19],

serviranno per creare un *Recommender System* collaborativo. I Sistemi di Raccomandazione si occupano di fornire agli utenti consigli personalizzati e di loro probabile interesse riguardo un set di oggetti appartenenti ad uno specifico dominio (e.g. ristoranti, cinema, pub, ecc.), a partire dai dati di cui si dispone: informzioni su utenti e oggetti.

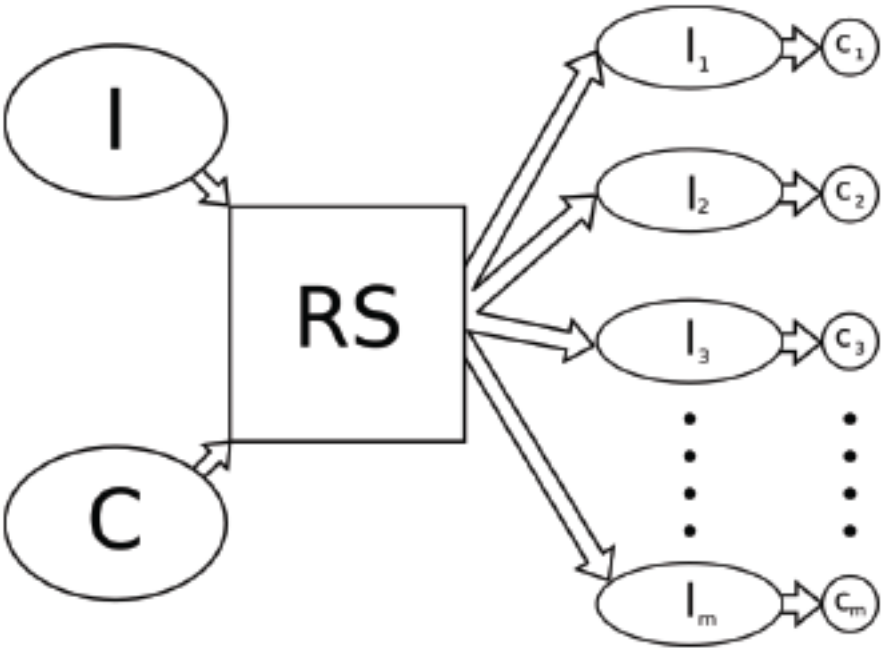


Figura 1.6: Un Recommender System elabora i dati sugli utenti C e gli oggetti I che raccomanda, per generare insieme di oggetti personalizzati per ciascun utente

Una definizione formale del problema delle raccomandazioni può essere espressa in questi termini: Sia C l'insieme degli $m \in N$ utenti del sistema, ed I l'insieme degli $n \in N$ oggetti (luoghi, cinema, parchi, ecc.) che possono essere raccomandati. La cardinalità degli insiemi C ed I può essere molto alta, ed i valori di m ed n possono essere dell'ordine di grandezza di centinaia di migliaia di unità. Si definisce inoltre la *funzione di utilità* $u : C \times I \rightarrow R$. Essa misura quanto un dato oggetto i sia utile ad un determinato utente c. R è un insieme totalmente ordinato, ed i suoi elementi rappresentano quantitativamente l'utilità di un'oggetto per un utente. L'*utilità* di un oggetto è tipicamente rappresentata

da una valutazione (ad esempio numerica) che un utente ha assegnato o assegnerebbe ad un oggetto.
Date queste definizioni, si vuole scegliere per ciascun utente $c \in C$, l'oggetto $i' \in I$ che massimizza la *funzione di utilità* per l'utente. Ovvero, formalmente:

$$\forall c \in C, i'c = \arg\max_{i \in I} u(c, i)$$

Il problema centrale delle raccomandazioni sta nel fatto che la funzione u non è definita su tutto lo spazio $C \times I$, ovvero è difficile che l'utente abbia valutato tutti gli oggetti nel dominio I, pertanto il Recommender System deve essere in grado di stimare i valori della *funzione di utilità* u anche nella parte dello spazio dei dati in cui non è definita, estrapolandoli a partire dai punti di $C \times I$ in cui essa è nota: questo è anche il compito più problematico. In altre parole l'obiettivo è fare una predizione sul voto che un determinato utente darebbe ad un oggetto del sistema che non ha ancora valutato.
Per quanto riguarda il progetto, il Recommender System servirà per segnalare dinamicamente sulla mappa dei guidatori i POI che con grande probabilità saranno di loro interesse.

Capitolo 2

Data Stream Management System (DSMS)

L'obbiettivo di questo capitolo è descrivere in dettaglio le caratteristiche principali dei moderni Data Stream Management System, facendo chiarezza sulle principali differenze rispetto ai DBMS classici, sulle caratteristiche delle query, dei dati e i contesti applicativi in cui sono applicati.

2.1 Caratteristiche generali

2.1.1 DBMS vs DSMS

In questi ultimi anni si sono sviluppate molte applicazioni *data-intensive* [13] in cui il dato deve essere gestito in maniera differente rispetto alla normale gestione prevista dai tradizionali DBMS. La prima grande differenza risiede nel fatto che in queste applicazioni si devono gestire stream di dati continui anzichè dati 'statici'. Uno stream di dati è una sequenza di oggetti (tuple) real-time, continui e ordinati dal *tempo di arrivo* o dal *timestamp* la cui dimensione totale è potenzialmente infinita. Date queste caratteristiche, le query non possono essere valutare sull'intero dataset ma solo su una certa *finestra temporale*. Generalmente nei DBMS i dati sono 'permanenti' con poche operazioni di inserimento/cancellazione/aggiornamento e le query su di essi sono molto complesse ed eseguite ad-hoc (*one-shot*) sull'intero dataset. In questa nuova classe di applicazione invece i dati sono 'transienti', ovvero le operazioni di inserimento e aggiornamento

su di essi sono molto frequenti, e le query richieste sono continue nel tempo ed eseguite su un sottoinsieme dell'intero dataset. Questi aspetti aprono un nuovo problema nel campo della ricerca.

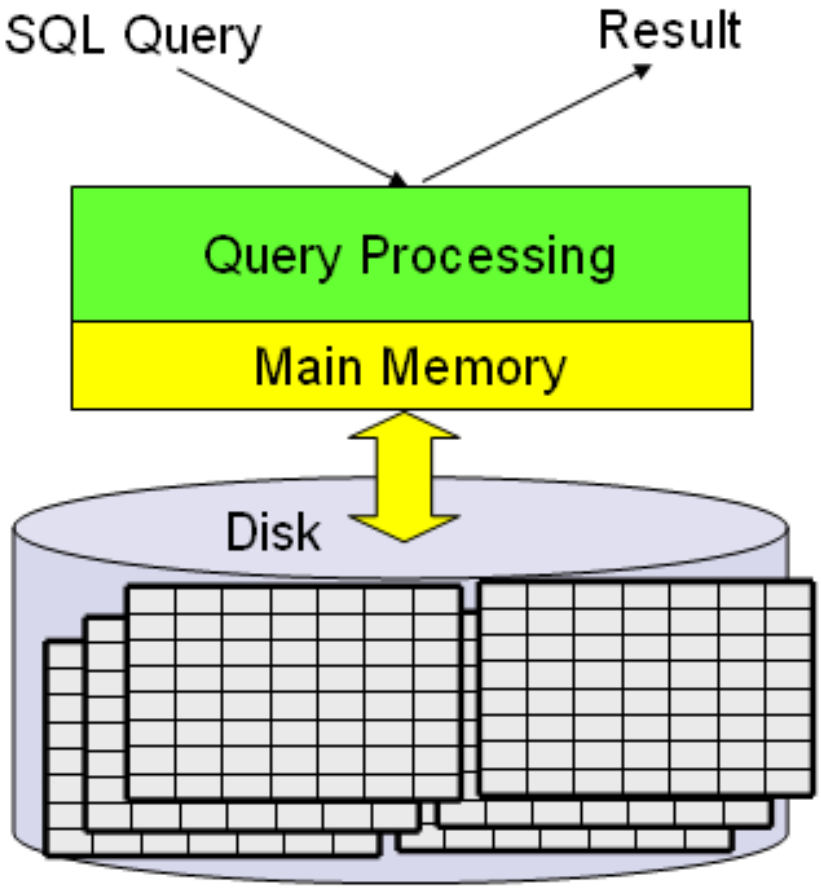


Figura 2.1: Principio di funzionamento di un DBMS

Le principali differenze sono riportate nella tabella 2.1.
Un esempio di queste applicazioni sono l'analisi finanziaria, il monitoraggio delle reti, la sicurezza, le sensor network, l'analisi delle richieste web, ecc. In queste applicazioni risulta estremamente inefficiente inserire i dati in arrivo in un normale DBMS ed eseguire delle normali query su di essi. Questo perchè i normali DBMS non sono stati progettati per gestire stream di dati e non offrono in maniera diretta delle funzionalità per gestire query continue. Inoltre, per garantire le performance nell'esecuzione delle query sugli stream di dati, devono essere previsti meccanismi di *approximation* (possibilità di fornire risposte ap-

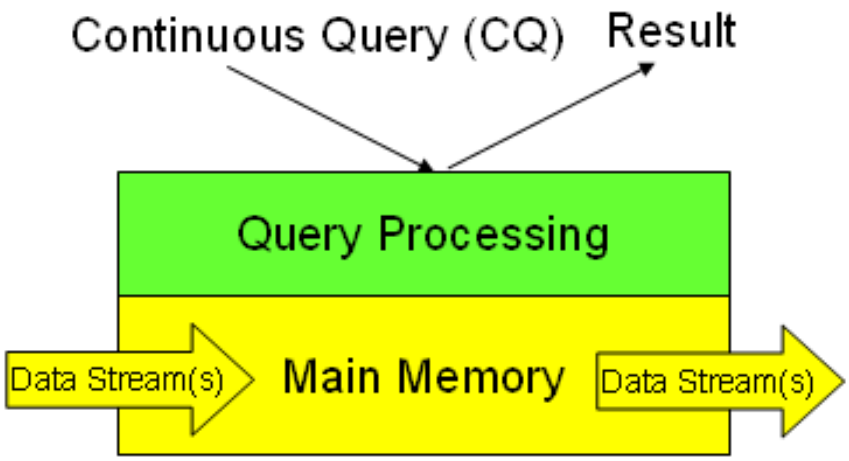


Figura 2.2: Principio di funzionamento di un DSMS

prossimate) e *adaptivity* (cambiamento dinamico del query plan al cambiare delle caratteristiche dello stream in input), aspetti che sono completamente opposti alle caratteristiche dei DBMS tradizionali che prevedono query plan stabili e risposte precise.
Le query approssimate vengono utilizzate quando per ottenere risposte precise è richiesto troppo tempo e quindi la risposta perde di importanza. Spesso si preferisce ottenere risposte meno precise ma fornite in un tempo più rapido e queste tipologia di query sono spesso utilizzate in contesti hard real-time. L'adaptivity è richiesto in quanto i dati arrivano online e il sistema non ha alcun controllo su di essi, quindi le caratteristiche dei dati possono cambiare e di conseguenza mantenere un query plan statico può risultare inefficiente. Questa tecnica di ottimizzazione è divisa generalmente in due step. Il primo si occupa di effettuare a runtime delle statistiche, come ad esempio il tasso di arrivo dei dati, la selettività degli operatori attuali, ecc. Il secondo step è cercare una sostituzione al query plan attuale con uno più efficiente, mantenendo la stessa semantica della query.
Il DSMS deve inoltre poter gestire l'esecuzione di più query contemporaneamente (*shared execution*) in maniera scalabile ed efficiente, per cui devono essere previsti meccanismi e politiche per l'accesso condiviso ai dati e ottimizzazione di query concorrenti. Ottimizzare delle query concorrenti significa evitare di eseguire più volte lo stesso calcolo sugli stessi dati, memorizzando ad esempio

Tabella 2.1: DBMS versus DSMS

Data Base MS	**Data Stream MS**
- Relazioni persistenti	- Stream transienti (online analysis)
- Query one-shot molto complesse	- Query continue piuttosto semplici (CQs)
- Accesso prevalentemente random	- Accesso prevalentemente sequenziale
- Spazio sul disco 'illimitato'	- Memoria primaria limitata
- Importanza del dato presente	- Importanza dei dati storici
- Nessun servizio real-time	- Requisiti hard real-time
- Operazioni di update poco frequenti	- Frequenze di arrivo anche nell'ordine di più Gigabytes
- Dati precisi e corretti	- Possibilità di dati danneggiati o mancanti
- Query plan statico determinato dal query processor	- Cambiamenti impredicibili nello stream di dati in arrivo e quindi necessità di query plan dinamici
- Query con risposte esatte	- Necessità di fornire risposte approssimate in breve tempo
- Umano Attivo, Database Passivo (HADP)	- Database Attivo, Umano Passivo (DAHP)
- Query sequenziali	- Più query eseguite contemporaneamente

dei risultati intermedi (chiamati *synopsis*) che possono essere riutilizzati da altri operatori di altre query, in modo da ottenere il risultato finale più rapidamente evitando di rieseguire più volte gli stessi calcoli.
I normali DBMS possono essere visti come un modello *HADP (Human Active, Database Passive)*, ovvero modelli in cui il database è un repository 'passivo' di informazioni su cui l'operatore esegue delle query e transazioni. Inoltre i *trigger* e gli *alerters* sono di minore importanza rispetto ai dati; il risultato più evidente di questa assunzione è che nessun database oggi disponibile risulta scalabile quando vengono definiti un grande numero di trigger, ossia superata una certa soglia di trigger definiti, il sistema degrada rapidamente le sue prestazioni. Nei sistemi in cui i dati sono a flusso, le funzionalità sono nella maggior parte dei casi basate proprio su trigger. I DSMS sono orientati secondo una filosofia *DAHP (Database Active, Human Passive)*, ovvero modelli in cui il database non acquisisce i dati da transazioni umane ma in automatico da sorgenti esterne. Gli

utenti non eseguono delle query sui dati ma vengono notificati in automatico dal sistema in caso di necessità.

2.1.2 Data stream

Un data stream è una sequenza di *data item* potenzialmente infinita la cui frequenza di arrivo può variare nel tempo. La sequenza è ordinata dal *timestamp* che indica il tempo logico di inserimento del data item nello stream. Ciascun data item nello stream può essere visto come una tupla con lo schema $< V, TS >$, dove V sono gli attributi che descrivono la tupla e TS è il timestamp generato dal sistema. Ad alto livello, si possono distinguere due tipi di data stream:

- **Transactional Data Streams**: Questa tipologia di data stream è rappresentata dalla memorizzazione (log) delle interazioni tra entità. Esempi di applicazioni con questa tipologia di dati sono:
  - *Siti Web*: In molti siti web le interazioni tra gli utenti e il sito sono memorizzate all'interno dei log allo scopo di monitorare le performance ed eseguire personalizzazioni. Le informazioni su queste interazioni sono continuamente inserite all'interno dei log sottoforma di *log entries*. I log vengono anche usati per rilevare intrusioni o attacchi.
  - *Banche*: Le banche hanno interessi nel monitorare e memorizzare le transazioni effettuate con le carte di credito allo scopo di rilevare anomalie o utilizzi fraudolenti. Queste informazioni vengono continuamente inserite all'interno dei log e rappresentano uno stream continuo di dati transazionali.
- **Measurement Data Streams**: Questo tipo di data stream è prodotto dal monitoraggio in real-time dello stato di entità di interesse. Esempi di applicazioni di questo tipo sono:
  - *Sensori in ambienti reali*: Una grande area di applicazioni riguarda il monitoraggio dei valori provenienti dai sensori posizioni in ambienti o entità reali. Questi sensori generano un grosso flusso di dati. Possono essere applicati ovunque e permettono il controllo, ad esempio, del traffico urbano, dello spostamento di soldati su un certo territorio, della temperatura di contenitori di gas, ecc.

- *Sensori in ambienti virtuali*: Alcuni contesti richiedono il monitoraggio di sistemi software distribuiti allo scopo di identificare colli di bottiglia o rilevare eventuali attacchi alla rete. I dati provenienti dai vari componenti possono essere di varia natura (utilizzo CPU, throughput di disco, rete, ecc) e generano uno stream di dati.

Una frequenza di arrivo dei dati molto elevata genera molto stress sulla rete e sulle risorse computazionali e può risultare difficile eseguire i compiti di:

- *Transmit (T)*: Trasferire i dati in input al programma.
- *Compute (C)*: Elaborare i dati in arrivo.
- *Store (S)*: Memorizzare in memoria principale la finestra temporale su cui effettuare le operazioni.

Questi aspetti devono essere considerati in fase di progettazione di un DSMS. Ad esempio, quando i dati vengono trasmessi, se il link di comunicazione è lento o i dati vengono danneggiati, si potrebbe avere un delay nella trasmissione di tali dati all'applicazione e di conseguenza un ritardo nelle risposte, per cui controllare solamente che i dati arrivino correttamente all'applicazione è sbagliato. Se la potenza computazionale è limitata o se la complessità dell'applicazione è alta sarà necessario molto tempo per ottenere le risposte, per cui controllare solamente la correttezza dei risultati rappresenta un altro errore. Infine, considerare di poter salvare tutti i dati di cui si ha bisogno rappresenta un altro errore, in quanto la memoria principale è generalmente limitata. Per questi motivi l'analisi *TCS* deve essere eseguita con grande attenzione. Semplificare troppo questi aspetti può voler dire non raggiungere gli obbiettivi di performance stabiliti. Ci sono due filoni di ricerca che hanno prodotto nuove sfide nel campo delle infrastrutture TCS:

- *Necessità di automatizzare le operazioni di memorizzazione dei dati provenienti da stream*: Questa necessità è nata molti anni fa quando le reti iniziarono a venire sfruttate nei sistemi che creavano un massiccio data stream: operazioni di transazioni bancarie, immagini ad alta risoluzione provenienti dai satelliti, informazioni metereologiche provenienti dai radar, misurazioni delle radiazioni atmosferiche, ecc.
  Internet è un sistema di rete *general-purpose* fortemente distribuito in cui

le sorgenti di dati e i consumatori di dati sono sparsi tra milioni di utenti. La frequenza delle transazioni web è aumentata esponenzialmente nel corso del tempo e genera numerosi streams: browser clicks, query degli utenti, log del traffico IP, dati delle email, web server, peer-to-peer downloads, ecc. Per questo motivo riuscire ad acquisire e gestire questi stream di dati in maniera automatica risulta fondamentale.

- *Necessità di eseguire in real-time analisi avanzate sui dati provenienti da stream*: Nei sistemi tradizionali di acquisizione dei dati le query eseguite in real-time sono relativamente semplici mentre quelle più complesse, come l'analisi dei trend o le previsioni future, sono generalmente eseguite offline nel warehouse. I sistemi più moderni, come ad esempio le applicazioni di monitoring dei dati provenienti da sensori atmosferici, astronomici, di rete e finanziari, hanno invece la nacessità di eseguire delle query complesse in hard real-time sui dati in arrivo in modo da rilevare i cambiamenti e le situazioni anomale nel più breve tempo possibile. In questi sistemi i vincoli temporali sono di fondamentale importanza e non rispettarli potrebbe significare grosse perdite economiche e, nei casi più estremi, pericolo per la vita delle persone.

La comunità scientifica in questi anni si è focalizza principalmente sugli aspetti legati a fattori di scala relativi alla 'dimensione': manipolare efficientemente una grossa mole di dati con opportune strutture dati, scalare in un database con petabytes di dati, sintetizzare un grosso dataset, ecc. Meno attenzione viene invece rivolta agli aspetti legati alle performance e il benchmarking dei sistemi sottoposti a rapidi aggiornamenti con requisiti hard real-time.
Ci sono varie strade per sviluppare sistemi che rispettano i requisiti imposti dal TCS. Questi sistemi sono altamente sofisticati e i principi fondamentali ad alto livello per svilupparli sono:

- *Pensare in parallelo* sia per quanto riguarda il processing dei dati (C) sia per quanto riguarda lo storage dei dati (S).

- *Controllare il data rate tramite sampling o load shedding* in modo da non sovraccaricare il sistema e poter gestire i dati in arrivo.

- *Utilizzare delle strutture dati adatte* in base alle caratteristiche dei dati in arrivo e al tipo di interrogazioni in modo da gestire piu efficientemente i dati e migliorare le performance.
- *Analizzare i dati in maniera gerarchica* in modo da effettuare analisi più rapide ma meno dettagliare nei livelli più bassi e analisi più complesse ma che richiedono più tempo nei livelli più alti con pochi dati.

Inoltre, dato che le funzioni che devono essere calcolate sugli stream di dati sono solitamente complicate e non risolubili in maniera esatta e deterministica senza il salvataggio dell'intero input, gli algoritmi devono considerare la possibilità di fornire risposte approssimate o randomizzate.

Modelli di Data Stream

Nei modelli più diffusi di data stream, il flusso di dati ricevuto in input rappresenta gli aggiornamenti di un segnale sottostante discreto e a valori reali.
Sia $A[1...m]$ il vettore rappresentante il segnale discreto con valori in $\mathbb{R}$. Sia $\mathbf{a} = (a_1, ..., a_N)$ il flusso di dati in arrivo, dove $a_i = (j, ci)$ con $1 \leq i \leq N$, $1 \leq j \leq m$ e $c_i \in \mathbb{R}$. Il flusso $\mathbf{a}$ arriva al sistema di calcolo sequenzialmente e in ordine crescente (ovvero a_i precede a_h se e solo se $i < h$), e la ricezione dell'i-esimo elemento a_i implica il seguente aggiornamento del segnale: $A[j] = A[j] + c_i$. Si

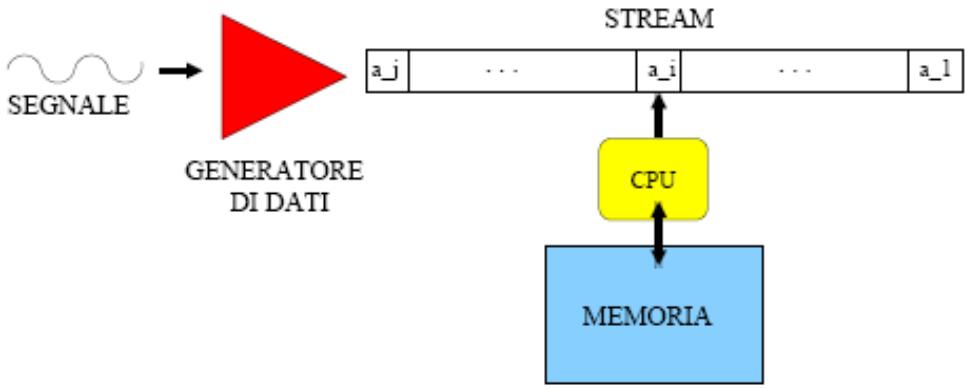


Figura 2.3: Esempio di sistema di calcolo per il data stream.

suppone che all'inizio dello stream, il vettore A sia nullo.

Si possono distinguere principalmente tre modelli di data stream, che si differenziano per i vincoli sugli elementi dello stream:

- *Time Series Model (TSM).* In questo modello $N = m$ e $a_i = (i, c_i)$ con $1 \leq i \leq m$, ovvero il valore di $A[i]$ viene aggiornato una sola volta da a_i, ovvero dall'i-esimo elemento dello stream. Questo modello è adatto in quei problemi in cui lo stream rappresenta l'evolversi temporale di un segnale, ad esempio il numero di pacchetti IP che transitano nell'unità di tempo in un link, il che implica $c_i = 1$.

- *Cash Register Model (CRM).* In questo modello $a_i = (j, c_i)$ e $c_i \geq 0$ per ogni valore di i, con $1 \leq i \leq N$ e $1 \leq j \leq m$. Non ci sono vincoli sull'ordine con cui vengono ricevuti gli aggiornamenti. Questo modello può essere utilizzato ad esempio per determinare gli indirizzi IP che inviano più pacchetti lungo un link: ogni posizione del segnale contiene il numero di pacchetti inviati da un certo IP, l'arrivo di un elemento dello stream indica la ricezione di un nuovo pacchetto e quindi il valore in una particolare posizione del segnale deve essere incrementato.

- *Turnstile Model (TM).* In questo caso non ci sono vincoli nè sull'ordine con cui vengono compiuti gli aggiornamenti nè sui c_i. Il nome deriva dai tornelli (*turnstile*) che tengono traccia delle persone che entrano o escono dalla metropolitana.

2.1.3 Query continue

Le query sui data stream hanno molto in comune con le query dei tradizionali DBMS. Tuttavia, ci sono due principali differenze se si vogliono gestire efficientemente i data stream. La prima differenza sta nel fatto che le tradizionali query dei DBMS sono *one-time queries* (o *one-shot queries*), ovvero vengono valutate ed eseguite una sola volta sull'intero data set e la risposta viene ritornata all'utente. Nei DSMS invece le query sono principalmente *continuos queries* (o *persistent queries*), ovvero vengono valutate ed eseguite continuamente sul data stream in arrivo. Ad ogni esecuzione della query continua vengono fornite nuove risposte sulla base dei dati che sono arrivati. Queste risposte possono essere memorizzate in opportuni storage oppure possono diventare a loro volta nuovi data stream che andranno in ingresso ad altre query.
La seconda distinzione è tra le *predefined queries* e le *ad hoc queries*. Le query

predefinite sono query che possono essere specificate prima dell'arrivo dei dati e sono generalmente anche continue. Le query ad hoc sono invece sottomesse online quando i dati sono già presenti e sono tipicamente one-shot. Le query ad hoc rendono più difficile la gestione dei dati perchè non essendo conosciute a priori è difficile effettuare operazioni di ottimizzazione.
Le query continue possono essere a loro volta classificate in due categorie sulla base del trigger che le fanno eseguire:

- *Change-Based*: La query viene eseguita quando si verifica un cambiamento nei dati in arrivo.

- *Timer-Based*: La query viene eseguita ad intervalli di tempo regolari specificati dall'utente che sottomette la query.

In una query continua si può anche eventualmente speficare una *stop condition* che specifica la condizioni necessarie per fermare la query. Le stop condition possono essere a loro volta time-based, se la condizione di stop è specificata da un parametro temporale (es: ferma la query se sono passati più di 5 giorni), o event-based, se la condizione di stop è specificata dal verificarsi di un evento (es: ferma la query se il prezzo sale più del 5% rispetto al suo valore originario).
Formalmente una query continua può essere specificata come una tripla $(Q, T_{cq}, Stop)$, in cui rispettivamente Q rappresenta la query da eseguire scritta in un linguaggio SQL-like, T_{cq} rappresenta il trigger che la fa eseguire e $Stop$ rappresenta la stop condition. Il risultato della query Q sul data set S_i può essere indicato con $Q(S_i)$. Il risultato di una query continua può essere definito come la sequenza delle risposte $\{Q(S_1), Q(S_2), ..., Q(S_n)\}$ ottenute eseguendo la query Q sui vari data set S_i, $1 \leq i \leq n$, il cui trigger di $Q(S_i)$ è $T_{cq} \wedge \neg Stop$.

Le caratteristiche particolari del modello dei dati e delle query continue richiedono al data stream management system le seguenti caratteristiche:

- Il modello dei dati e la semantica delle query devono prevedere delle operazioni time-based (es: finestre temporali) e order-based.

- L'incapacità di poter memorizzare l'intero data set richiede alle query la possibilità di fornire risposte approssimate o incomplete interrogando delle strutture dati che contengono un 'riassunto' di un certo data set. In letteratura queste strutture dati sono note come *synopses* o *digests*.

- Gli streaming query plans non devono usare degli operatori bloccanti [15] che consumano tutto l'input prima di fornire alcuna risposta.
- Dati i vincoli di performance e memoria primaria limitata non deve essere possibile, per una stessa query, accedere più volte sugli stessi dati (*backtracking*). Gli algoritmi di interrogazione devono prevedere una sola passata sui dati.
- Le applicazioni che monitorizzano gli stream in real-time devono reagire velocemente a dati anomali o inusuali.
- Le query continue eseguite per lungo tempo possono incontrare cambiamenti nelle condizioni del sistema o dello stream di input e quindi devono essere previsti meccanismi di adattamento alle variazioni in modo da garantire bassi tempi di risposta.
- L'esecuzione condivisa di più query deve essere sempre prevista in modo da garantire la scalabilità del sistema.

Per quanto riguarda i linguaggi di interrogazione vengono utilizzati varianti del linguaggio di interrogazione relazione SQL. Anche se la tipologia di dati e di applicazioni potrebbe portare ad una scelta progettuale verso un linguaggio nuovo di interrogazione, la scelta di estendere SQL è sicuramente condivisibile in quanto rende la curva di apprendimento del nuovo linguaggio meno ripida.
La soluzione migliore consiste nell'estendere il linguaggio SQL con una serie di operatori atti a permettere operazioni su 'finestre' di dati provenienti da una sorgente. Solitamente, infatti, il flusso dei dati di una certa sorgente viene suddiviso in finestre delimitate o temporalmente oppure da un numero predefinito di tuple. Una volta che una finestra è disponibile, i trigger registrati dall'utente per quella sorgente vengono invocati e, quando le operazioni su quella finestra sono state tutte eseguite, le tuple in essa contenute vengono eliminate. Alcuni sistemi fanno eccezzione (come previsto nel DSMS di PEGASUS) in quanto non prevedono la cancellazioe dei dati 'consumati'. Questi sistemi memorizzano su memorie di massa i dati consumati e dedicano particolare attenzione all'analisi dei dati storici: l'approccio generalmente scelto è quello di creare repository distribuiti di tipo Warehouse per supportare efficientemente eventuali analisi OLAP da parte degli utenti del sistema.
Le specifiche per un linguaggio di interrogazione per DSMS sono due: l'efficienza

e l'espressività. L'efficienza è assolutamente necessaria a causa della particolare natura dei dati generati dalle sorgenti. Essi sono infatti potenzialmente illimitati ed il loro arrivo imprevedibile a priori. Inoltre, l'efficienza è necessaria non soltanto in fase di arrivo ed elaborazione dei dati, ma anche in fase di interrogazione. Infine, poiché le applicazioni che usano dati a flusso sono molto differenti fra loro, in generale è molto più significativo un sistema che riesca ad adattarsi alle particolari condizioni di esecuzione oppure alla natura dei dati. Qui entra in gioco l'altro parametro importante: l'espressività. Grossa parte dell'espressività di un sistema è garantita dal linguaggio adottato per esprimere le interrogazioni (queries) e per elaborare i flussi di dati provenienti dalle sorgenti. Più formalmente potremmo definire l'espressività di un sistema come la facilità per i suoi utenti di esprimere le interrogazioni. Più un sistema è espressivo e più dovrebbe essere semplice esprimere la maggior parte delle queries.

2.2 Scenari di applicazione

In questa sezione verranno descritti i principali contesti applicativi nei quali i DSMS si inseriscono. Verranno inoltre approfondite le caratteristiche che il DSMS dovrà avere sulla base dell'applicazione in esame e la tipologia delle query che potranno essere sottomesse.

2.2.1 Reti di sensori

Le reti di sensori [10] sono architetture hardware e software appositamente ideate per implementare un supporto adeguato al problema di inoltrare i dati generati da un insieme di sensori facenti parte di una rete solitamente wireless, verso una stazione base detta *sink*. Il sink ha il compito di processare opportunamente i dati ricevuti e di memorizzarli oppure di instradarli verso un'altra macchina o rete. I recenti progressi tecnologici nei sistemi elettronici e micro-elettronici e nelle comunicazioni wireless hanno permesso lo sviluppo di Nodi Sensore (detti anche sensor node) molto piccoli e a basso assorbimento di energia, dai costi abbastanza contenuti.
Nei sistemi di dati a flusso (DSMS) e nelle reti di sensori (SN), interviene il concetto di sensore o sorgente di dati a flusso, o sorgente di dati continui. L'aggettivo

continuo sta ad indicare la proprietà della sorgente di dati circa la generazione ininterrotta di misurazioni potenzialmente illimitata nel tempo. Volendo dare una definizione di sensore, possiamo definire sensore un qualsiasi dispositivo capace di misurare e trasmettere informazioni dell'ambiente (environment) in cui è stato dislocato. Solitamente un sensore è specializzato nella misurazione di una particolare grandezza, come ad esempio: Temperatura, Pressione, Posizione GPS, Immagini, ecc.
Le reti di sensori si stanno diffondendo in maniera capillare e stanno diventando un ambito di studio molto promettente in quanto esistono molteplici campi di applicazione per queste tecnologie. Una prima suddivisione può essere fatta in base alla finalità della rete. Di seguito si riportano quattro macro-settori in cui è possibile dividere il mondo delle WSN:

- **Monitoraggio**: Questo tipo di rete viene utilizzata per tracciare in maniera continuativa una certa grandezza. La sorgente da monitorare può essere un singolo sensore o una sottorete da cui proviene un aggregato di dati. Una rete di questo genere richiede un campionamento solitamente isocrono e fisso con un consumo energetico considerevole.
  In questa categoria rientrano vari contesti applicativi tra cui: il controllo del traffico in una grande città, la raccolta di dati da un'area colpita da un terremoto, la video-sorveglianza di un quartiere, l'analisi degli spostamenti delle merci in una regione o stato, ecc.

- **Riconoscimento di eventi**: Un'altro importante scenario applicativo risulta essere l'*event detection*: la rete deve accorgersi di situazioni di 'allarme', ossia di quando una determinata grandezza esce dai livelli stabiliti. Questo monitoraggio è potenzialmente meno pesante del precedente a livello energetico, in quanto il nodo entra in uno stato attivo solo in casi eccezionali.

- **Classificazione di oggetti**: L'obbiettivo di queste reti è il riconoscimento di alcune grandezze tra un insieme di prototipi noti. Questo implica un carico computazionale superiore agli altri tipi di rete.

- **Tracciamento di oggetti**: In questo caso la rete funge da sistema di sorveglianza, riconoscendo e stimando la posizione di alcuni oggetti in una determinata area geografica.

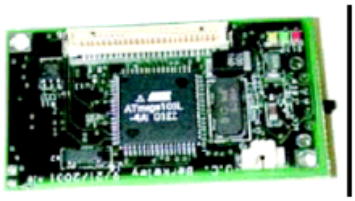
Una caratteristica opzionale ma sicuramente desiderabile di una Sensor Network è la possibilità che i nodi sensori collaborino e comunichino tra loro. I nodi sono infatti solitamente provvisti di un processore on-board capace di effettuare semplici elaborazioni. Ciascun nodo, quindi, invece di inviare dati 'grezzi' ai nodi responsabili di raccogliere i dati, può effettuare delle operazioni di aggregazione o pulizia dei dati e solo successivamente o su richiesta trasmetterli già processati. In questo contesto l'utilizzo di un data stream management system risulta fondamentale al fine di memorizzare ed analizzare il data stream in arrivo dai sensori. Le caratteristiche principali che un DSMS sviluppato per una rete di sensori deve avere è la sua capacità di minimizzare la comunicazioni tra i nodi sensore e massimizzare la durata delle batterie, in quanto generalmente le sensor network hanno le seguenti limitazioni:

- **Comunicazione**: la banda delle connessioni wireless, adottate dai nodi sensore, è solitamente limitata a poche centinaia di Kbps. La qualità del servizio, inoltre, è molto bassa perché affetta da rumore, da alta latenza e con forte perdita di pacchetti.

- **Consumo di energia**: i nodi sensore di una sensor network hanno sorgenti di energia limitate. Il consumo di energia, infatti, è uno dei più importanti parametri da considerare durante la progettazione di una rete di sensori. Le attuali batterie di ridotte dimensioni forniscono circa 3000 mAh di capacità. Considerando il sensore MICA riprodotto in figura 2.4, una batteria di 3000 mAh lo riesce ad alimentare per un anno in stato idle e per una settimana in stato di trasmissione.

- **Incertezza nelle letture dei sensori**: i segnali misurati dai sensori fisici hanno delle limitazioni dovute alle limitazioni del sensore stesso ed anche al rumore dell'ambiente circostante. I guasti o le posizioni errate dei sensori potrebbero portare ugualmente a valori errati o forte rumore nell'acquisizione.

L'incertezza del dato letto da un sensore può essere descritta da una funzione di distribuzione probabilistica che aiuti ad interpretare il singolo dato. Ad esempio, considerando un sensore che misuri la temperatura, se una delle sue letture è 25 gradi, si può calcolare la probabilità che il dato letto sia corretto. Se assummessimo che l'errore introdotto dal sensore avesse una distribuzione



Processor	4Mhz, 8bit, MCU
Memory	128KB Flash, 4KB RAM, 4KB ROM
External Flash	521KB
Radio	916Mhz Radio (RF Monolithics)
Communication Range	200 ft
Data Rate	50 Kbits/sec
Transmit Current	12 mA
Receive Current	1.8 mA
Sleep Current	5 uA

Figura 2.4: Il sensore MICA sviluppato alla Berkeley University.

Gaussiana con una deviazione standard di σ gradi, allora potremmo calcolare la possibilita che la vera temperatura T si trovi nell'intervallo [T1, T2]. Nel contesto di una rete di sensori un utente dovrebbe poter formulare una query che ritorni tutte le temperature i cui valori reali ricadono nell'intervallo [T1, T2] con una data probabilità. Inoltre, se si hanno più sensori che misurano contemporaneamente la medesima grandezza, si può costruire un 'super nodo' virtuale che fornisce i valori di temperatura con molta meno varianza dei singoli sensori. Il DSMS deve inoltre massimizzare il QoS verso le applicazioni finali, ovvero deve minimizzare la latenza, massimizzare il throughput e migliorare la qualità dei risultati. Questi aspetti vanno spesso in contrasto con gli obbiettivi primari di una sensor network che cerca di minimizzare il consumo di batteria e minimizzare le comunicazioni. In genere la scelta su come configurare il DSMS e la SN viene lasciata al progettista che sceglie, in base al QoS richiesto dalle applicazioni finali, il tradeoff migliore tra consumo di potenza, latenza e qualità delle risposte fornite dal DSMS e dalla SN. Generalmente più la latenza richiesta è bassa e la qualità delle risposte è alta maggiore sarà il consumo di energia da parte della SN perchè ai nodi sensori sarà richiesto un maggiore carico computazionale. Viceversa, se i vincoli legati al QoS sono meno stringenti, ai nodi della SN verranno richieste meno elaborazioni e un minor numero di comunicazioni e questo aumenta l'autonomia dei nodi sensore.

Nelle reti di sensori sono molto frequenti le query di aggregazione (AVG, MAX, MIN, ecc.). Un esempio è il seguente:

```
SELECT AVG(R.concentration)
FROM ChemicalSensor R
WHERE R.loc IN region
HAVING AVG(R.concentration) > T
DURATION (now, now+3600)
EVERY 10
```

In questa tipologia di interrogazioni le ottimizzazioni sul query plan mirano a minimizzare le comunicazioni e la computazione. La query raccoglie i dati provenienti da una regione di sensori (*communication compotenets*) e li invia ad un nodo centrale per le operazioni di aggregazione (*computation component*). Esistono diverse tecniche di ottimizzazione che sono:

- *Direct delivery*: Questo è lo schema piu semplice. Ciascun nodo sorgente invia i propri record al nodo leader attraverso un routing multi-hop. La computazione è interamente effettuata dal leader una volta ricevuti tutti i record.

- *Packet merging*: Nelle comunicazioni wireless è meno dispendioso inviare un unico grosso pacchetto piuttosto che tanti piccoli pacchetti. Questo avviene perchè su ciascun pacchetto bisogna pagare un costo aggiuntivo legato all'header. In questo senso, i sensori presenti in una certa regione possono eseguire un *merge* dei record e inviare al leader un unico pacchetto di dimensioni maggiori, diminuendo quindi i consumi e le comunicazioni globali.

- *Partial aggregation*: Per alcuni operazioni di aggregazione si possono mantenere dei risultati intermedi che vengono di volta in volta aggiornati con nuovi valori. Si possono quindi utilizzare dei nodi intermedi per svolgere alcune operazioni di aggregazione che verranno usate dal leader per costruire il risultato finale. Questo diminuisce il carico computazionale e le comunicazioni verso il nodo leader.

2.2.2 Analisi del traffico di rete

La crescente complessità delle reti odierne rende i servizi di analisi e di certificazione delle performance sempre più importanti. La molteplicità di protocolli, di interfacce e di servizi rende difficile misurare l'impatto che può avere sulle prestazioni del sistema l'aggiunta di traffico, di nuove applicazioni o di protocolli. In più, picchi eccessivi di traffico, o picchi di traffico anomalo, possono causare problemi di rete più o meno temporanei, e se non si ha la possibilità di ricreare ed isolare le condizioni che hanno causato il problema, risulta pressoché impossibile impedire che le condizioni di criticità si ripresentino in futuro. Tali considerazioni hanno creato la necessità di avere strumenti sofisticati che permettano di realizzare una dettagliata analisi ed una corretta calibrazione di una rete, prima che essa diventi operativa, così come strumenti che consentano una dettagliata analisi di prestazioni di una rete già esistente.
Attualmente, vi sono diversi strumenti in grado di analizzare il traffico di una rete [3]; essi possono essere basati sulla capacità di filtrare il traffico sulla base di indirizzi o protocolli, e di catturarne e decodificarne le trame. Tali strumenti, tuttavia, vengono prevalentemente utilizzati per la risoluzione di problemi di rete: sono sistemi cosiddetti **reattivi**. L'analisi del traffico consente ad un network manager di sapere la quantità e la qualità del traffico che viaggia sulla rete, e di evidenziare eventuali picchi di carico. Filtrando e catturando il traffico è inoltre possibile effettuare una analisi molto più profonda in grado di rivelare anche i più piccoli dettagli di ciò che avviene sulla rete.
Sebbene queste informazioni possano essere utilizzate per analizzare le prestazioni della rete, esse prendono in considerazione solo il traffico esistente. Per verificare realmente la capacità e le prestazioni della rete è necessario introdurre differenti tipi e quantità di traffico in modo da analizzare le prestazioni del sistema in condizioni limite. Questo modo di concepire il network management può essere definito **proattivo**, in quanto la misura viene effettuata stimolando opportunamente la rete.
I programmi per l'analisi del traffico di rete sono molteplici e si dividono in diverse categorie:

- **Programmi per la cattura ed il filtraggio dei pacchetti**: Comunemente anche chiamati 'sniffer'. Gli sniffer intercettano i singoli pacchetti decodificando le varie intestazioni di livello datalink, rete, trasporto e ap-

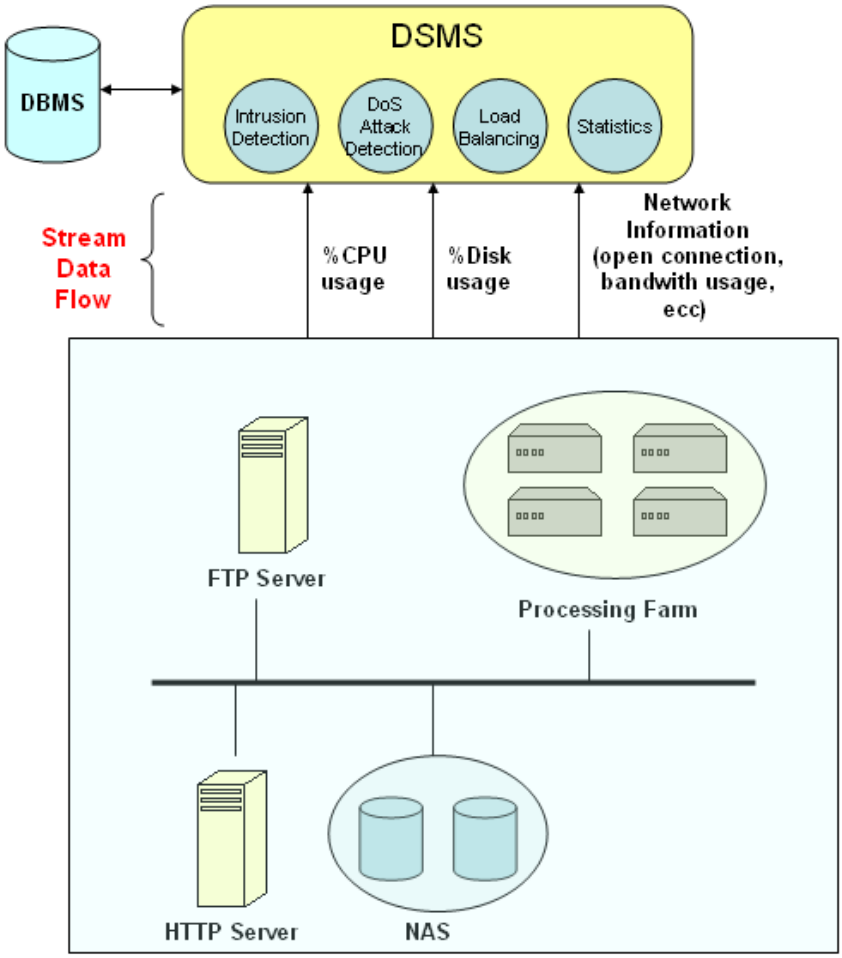


Figura 2.5: Data stream management system per l'analisi del traffico di rete.

plicativo. Inoltre possono offrire strumenti di analisi che analizzano, ad esempio, tutti i pacchetti di una connessione TCP per valutarne il comportamento o per ricostruire lo scambio di dati tra le applicazioni. Data la quantità di dati che possono circolare in una rete questi programmi generalmente presentano dei filtri in grado di rilevare solo i pacchetti di un certo tipo (IP, TCP, ICMP, ecc), origine, destinazione o lunghezza.

- **Programmi per il monitoraggio del traffico di rete**: Lo scopo primario di questi programmi è offrire delle statistiche in tempo reale sui pacchetti circolanti nella rete. Esempi di statistiche possono riguardare: il tipo di pacchetti, la loro dimensione, l'origine, la destinazione, ecc.

- **Programmi per la generazione di traffico**: Sono programmi in grado di generare del traffico nella rete di vario tipo. Molti usati per effettuare load testing e stress testing sulla rete.

Tutti questi programmi generano un grosso flusso di dati continuo che deve essere memorizzato e analizzato. L'utilizzo di un DSMS è quindi di fondamentale importanza. Occorre analizzare in real-time il traffico prodotto dalla rete per rilevare eventuali anomalie e situazioni critiche che possono compromettere il funzionamento e la sicurezza della rete (Congestioni, Denial Of Service, ecc). Analizzare i cambiamenti nella topologia della rete e nella distribuzione del carico può servire per rilevare le congestioni.
Negli attacchi di tipo Denial Of Service l'obbiettivo è saturare la *backlog queue* del nodo destinazione con richieste di attivazione di un servizio (TCP SYN settato) oltre la scadenza dei relativi timeout e non consentendo alla vittima di completare il 3-way handshake. In questo caso mantenere per ogni sorgente e per ogni destinazione il numero di step del 3-way handshake completati può essere utile per rilevare attacchi di tipo DoS. Il DSMS deve mantenere una *traffic matrices* che contiene per ogni sorgente e per ogni destinazione varie informazioni tra cui: la banda utilizzata, il numero di connessioni attive, i protocolli usati, la subnet mask, ecc. Questa matrice verrà utilizzata dalle query del sistema per rilevare anomalie o situazioni critiche.
Anche i dati storici hanno un ruolo importante in quanto possono venire sfruttati 'offline' per effettuare *capacity planning*, *provisioning* e *trend analysis*.

2.2.3 Analisi del traffico urbano

I trasporti rappresentano un settore nevralgico per lo sviluppo economico e sociale di ogni Nazione. È difficile immaginare una crescita economica forte, creatrice d'occupazione e di ricchezza, in assenza di un sistema di trasporto efficiente in grado di sfruttare pienamente il mercato interno e le possibilità offerte dalla globalizzazione degli scambi commerciali. L'aumento dei volumi di traffico, soprattutto stradali, ha portato ad un esacerbamento dei fenomeni di congestione, con conseguenti impatti negativi sull'ambiente, la qualità della vita e la sicurezza del trasporto. Il costo pagato giornalmente dalla collettività in termini di perdite di tempo, inquinamento e sicurezza è enorme, e il rischio è che la domanda di

trasporto in continua crescita possa rendere tale costo non più sostenibile. In questo scenario si inserisce anche il progetto PEGASUS.

L'analisi del traffico urbano ha come obbiettivo primario il miglioramento del traffico, diminuendo le congestioni, aumentando la sicurezza e riducendo i tempi di trasporto. Questo obbiettivo si può raggiungere solo tramite l'*Intelligent Transportation System*, ovvero inserendo le tecnologie informatiche e della telecomunicazione nelle infrastrutture dei trasporti e dei veicoli. L'ITS nasce dalla necessità di gestire i problemi causati dalla congestione del traffico attraverso una sinergia delle nuove tecniche informatiche per la simulazione, il controllo in tempo reale e le reti di comunicazione. Il traffico è in costante aumento in tutto il mondo, ed è il risultato di una crescente motorizzazione, urbanizzazione e della crescita demografica, nonché di un sostanziale cambiamento nella densità di popolazione. Il traffico riduce l'uso delle infrastrutture dei trasporti ed aumenta il tempo di percorrenza, l'inquinamento ed il consumo di carburante. Gli Intelligent Transportation Systems variano a seconda delle tecnologie applicate, dai sistemi di gestione base come navigatori satellitari, sistemi di controllo semaforici o rilevatori di velocità per applicazioni di monitoraggio applicate a sistemi di telecamere a circuito chiuso, fino alle applicazioni avanzate che integrano dati in tempo reale provenienti da varie fonti esterne, tipo informazioni meteorologiche, sistemi sghiacciamento dei ponti e simili. Oltre a queste, altre tecniche di previsione sono state sviluppate per permettere modellazioni avanzate e comparazioni con dati storici.
Le tecnologie informatiche e della telecomunicazione impiegate nell'ITS sono molteplici:

- **Wireless**: Varie forme di tecnologie di comunicazione wireless sono state proposte per i sistemi d'Intelligent Transportation. Le comunicazioni a corto raggio (sotto i 500 metri) possono essere effettuate utilizzando il protocollo IEEE 802.11 o lo standard *Dedicated Short Range Communications* promosso dalla Intelligent Transportation Society of America e dal dipartimento dei trasporti Americano.

- **Computer di bordo**: Le recenti innovazioni nel campo dell'elettronica per auto ha portato all'introduzione di piccoli processori sui veicoli che sono di fatto dei 'mini' computer. Sono in grado di elaborare le infor-

mazioni provenienti da vari sensori in real-time, collegarsi alla rete e inviare informazioni.

- **Sensori**: Possono essere posizionati sulle infrastrutture (strade, edifici, semafori, ecc) o sulle vetture e sono in grado di rilevare varie informazioni che potranno essere memorizzate ed elaborate.

- **Ricevitori GPS**: Sono dispositivi in grado di ricevere il segnale satellitare GPS (Global Positioning System) e quindi riescono a tenere traccia degli spostamenti e delle autovetture.

- **Cicli induttivi**: I cicli induttivi possono essere posizionati nell'asfalto per poter individuare i veicoli di passaggio attraverso il rilevamento del loro campo magnetico. I sensori più semplici si limitano a contare il numero di mezzi che, in un intervallo di tempo, passano sul sensore, mentre i più sofisticati riescono a valutarne anche la velocità, la lunghezza, il peso e la distanza tra loro. Le bobine possono essere piazzate in linea singola o multipla, e funzionano con veicoli veloci e lenti, o addirittura fermi.

- **Telecamere**: La misurazione del traffico attraverso l'uso di telecamere è un altro dei metodi utilizzati. Dal momento che i sistemi video non richiedono l'installazione di componenti nell'asfalto o ai bordi delle strade, questo genere di soluzione è definita 'non intrusiva'. I video catturati da videocamere in bianco e nero vengono passati ad un microprocessore che analizza le modifiche dell'immagine ripresa calcolandone il numero di veicoli. Le videocamere vengono solitamente montate su pali della luce o su strutture adiacenti le strade. La maggior parte di questi sistemi sfrutta le tecniche di apprendimento automatico e visione artificiale per riuscire a riconoscere i mezzi in movimento e la loro dimensione. Un tipico output è un elenco delle velocità dei mezzi, il loro conteggio e una misura d'occupazione della carreggiata. Alcuni sistemi forniscono ulteriori risultati, come la distanza tra i veicoli, l'andamento nel tempo, il riconoscimento di veicoli fermi e di quelli che si muovono contromano.

Le applicazioni finali dell'ITS possono essere:

- **Smart navigation**: La navigazione intelligente è una delle applicazioni principali dell'ITS e mira a migliorare il traffico urbano diminuendo le congestioni e offrendo percorsi alternativi.

- **Raccolta elettronica del pedaggio**: L'Electronic Toll Collection (ETC) permette ai veicoli di passare attraverso i caselli alla normale velocità del traffico, riducendo le code e raccogliendo in modo automatico il pedaggio. Originariamente i sistemi ETC venivano usati solo per il pedaggio, ultimamente sono stati inseriti anche per gestire le aree a traffico limitato nei centri cittadini, inserendo il cosiddetto 'pedaggio urbano'.

- **Sistemi di notifica degli stati di emergenza**: Questi sistemi migliorano la sicurezza del viaggiatore, ad esempio segnalando automaticamente gli incidenti ad una centrale operativa e di conseguenza richiedendo l'intervento dei soccorsi.

- **Gestione zone a traffico limitato**: Le zone a traffico limitato vengono usate principalmente nei centri urbani in cui il trasporto pubblico è un'alternativa alla guida. Gli autisti che entrano in una zona a traffico limitato sono soggetti al pagamento di un pedaggio superiore al costo del trasporto pubblico.

Le tecnologie viste sopra generano un grosso flusso di dati continuo che deve essere memorizzato ed analizzato in real-time. L'utilizzo di un DSMS risulta quindi essere un supporto fondamentale per ogni tipo di applicazione finale. Le caratteristiche che un DSMS deve avere per l'analisi del traffico sono state discusse nel capitolo precedente.

2.2.4 Analisi finanziarie

Un mercato finanziario è un *luogo economico* nel quale vengono scambiati strumenti finanziari di varia natura a medio o lungo termine. Per strumento finanziario si intende un qualsiasi mezzo di investimento finanziario (azione, obbligazione, titolo di mercato, ecc). Un mercato finanziario consente il trasferimento del risparmio dai soggetti che lo accumulano ai soggetti che lo richiedono (imprese). I soggetti che richiedono liquidità sono definiti 'soggetti in disavanzo finanziario' ed emettono strumenti finanziari (depositi bancari, azioni, buoni ordinari del tesoro (BOT), ecc.) che cedono ai soggetti in 'avanzo finanziario' in cambio di moneta. Lo scambio tra strumenti finanziari e moneta consente la redistribuzione dei rischi economici, perché vengono assunti in parte dagli ac-

quirenti degli strumenti finanziari. È possibile per questi ultimi cedere tali strumenti ad altri soggetti economici, scambiandoli nei mercati appositi. Esistono quindi mercati azionari, obbligazionari, dei derivati, delle opzioni, dei warrant, ecc., ognuno con proprie regole e proprie caratteristiche. Negli ultimi anni questi mercati sono cresciuti notevolmente e gestire la quantità di dati a disposizione risulta molto difficile.
Vengono sempre più richiesti strumenti e sistemi in grado analizzare i dati ed effettuare delle previsioni che permettano di visualizzare i trend futuri del mercato e i momenti migliori in cui acquistare/vendere. Questi sistemi devono analizzare una grossa mole di dati in arrivo dai vari mercati e rilevare condizioni favorevoli in cui poter acquistare. La latenza risulta essere un requisito fondamentale in quanto il mercato cambia rapidamente e il fattore 'tempo' può causare un guadagno o una perdita. Tutte queste caratteristiche richiedono l'utilizzo di un DSMS in grado di memorizzare le informazioni provenienti da più mercati azionari e in cui le query devono essere in grado di rilevare 'online' le variazioni di prezzo delle azioni al fine di scoprire delle correlazioni, identificare i trend e le situazioni favorevoli, prevedere i prezzi futuri e calcolare i rischi. *Traderbot* è un esempio di sistema di questo tipo e permette ai suoi utenti di porre delle query molto complesse come queste:

- Trova tutte le azioni il cui prezzo è compreso tra i 20$ e i 200$, dove negli ultimi 30 minuti la differenza tra il suo valore massimo e il valore minimo è maggiore del 3% rispetto al valore originario e dove negli ultimi 5 minuti il suo volume medio è aumentato più del 300%. (il volume misura la quantità di scambi registrati su un titolo o su un mercato. E' un parametro molto importante per l'analisi tecnica in quanto l'aumento dei volumi nella direzione del trend conferma la tendenza in atto)

- Trova tutte le azioni del NASDAQ che nel calcolo della media mobile hanno superato i 200 giorni (la media mobile viene usata per valutare il trend dei prezzi delle azioni) e in cui la *market capitalization* (che indica il valore di mercato del capitale azionario di una società, derivante dal corso delle sue azioni moltiplicato per il numero di azioni di capitale emesse dalla società) è maggiore di 5$ bilioni e che oggi ha guadagnato tra il 2% e il 5%.

2.2.5 Analisi dei log

L'analisi dei log è una delle attività principali in ambito web e sistemistico e può servire per diversi scopi:

- *Intrusion Detection*: Per intrusion detection si intende quel processo di scoperta, analisi e reporting delle attività non autorizzate che possono violare la confidenzialità, l'integrità e la riservatezza dei dati. La grande diffusione delle reti e di Internet come mezzi di accesso ad informazioni e servizi rende sempre più critico, per qualunque azienda, la protezione delle risorse da accessi non autorizzati e/o attacchi mirati all'acquisizione di privilegi non garantiti. Questo problema diventa di vitale importanza per l'economia di un'azienda, dal momento che quest'ultima ha bisogno di amministrare notevoli quantità di informazioni mantenendone segretezza, integrità e disponibilità.
  L'analisi dei log è di fondamentale importanza nel processo di intrusion detection sia nel caso ci siano stati dei tentativi di intrusione, che nel caso si sia verificato un incidente informatico. I log memorizzano record contenenti informazioni sul timestamp, sul protocollo utilizzato, sull'indirizzo IP sorgente e di destinazione e sulle porte utilizzate da qualsiasi attività anomala monitorata. Possono anche essere contenuti dati aggiuntivi come una descrizione testuale dell'evento o il numero della regola che ha generato l'allarme.
  Se analizzati correttamente, i log dei firewall possono evidenziare attacchi di Denial of Service (DoS) o altre attività sospette. Tuttavia, poche aziende analizzano questi log e anche quelle che lo fanno, lo fanno poco frequentemente. Quindi, molti tentativi di attacco passano inosservati, dando agli hacker un'elevata probabilità di superare la barriera rappresentata dai firewall. La firewall log analysis è un'operazione complessa che richiede il controllo simultaneo di migliaia di eventi.
  Inoltre, di estrema importanza è la capacità di reazione di fronte ad un'intrusione. Intervenire tempestivamente quando si identifica un'intrusione, spesso è essenziale per evitare che l'attaccante possa alterare o eliminare i log.

- *Profiling degli utenti*: Per profiling si intende l'attività per mezzo della quale una serie complessa di dati relativi ad utenti/clienti viene elaborata

da specifici programmi per generare alla fine la segmentazione della propria utenza in gruppi omogenei di comportamento. I dati che possono essere presi in considerazione per la profilazione sono molteplici: la serie delle scelte di navigazione effettuate sul sito in esame dagli utenti identificati, la dichiarazione esplicita di preferenze e interessi ottenuta tramite procedure di registrazione o sondaggi, la raccolta di dati demografici, ecc.
Quando un visitatore accede al sito lascia una traccia del suo passaggio in un archivio (log) che registra tutte le richieste effettuate a quel determinato sito ed in particolare l'ora, indirizzo IP utente, oggetto della richiesta (pagina web, immagine, video), browser utilizzato. I log di questo tipo possono essere usati per capire, dato un certo utente, quali sono le sue pagine preferite e a quale ora del giorno accede. Queste informazioni possono essere utilizzate, tramite un processo di Data Mining, per diversi scopi, come ad esempio capire i gusti degli utenti e potergli fare offerte mirate o sconti particolari, rinnovare la fidelizzazione di clienti che stanno per abbandonarci, valutare le pagine più visitate, ecc.
Un altro dato importante e relativo ai visitatori è il 'tempo medio' di permanenza sul sito. Se i contenuti sono molti ma il tempo trascorso on-line è basso (pochi secondi), probabilmente andrà riorganizzato il layout grafico per rendere più usabile e facile la lettura del testo sullo schermo.
Eseguire un corretto profiling degli utenti aiuta un azienda o sito a crescere e migliorarsi, oltre che aumentare le vendite e la fidelizzazione dei clienti.

- *Forecasting*: L'analisi storica dei log permette di effettuare previsioni su quello che sarà l'andamento futuro delle richieste. Per poter fare previsioni in modo accurato il primo presupposto è che il passato sia un buon predittore del futuro. Se determinati clienti sono in condizione di insolvibilità in seguito a determinati comportamenti, allora anche clienti simili seguiranno un comportamento analogo in futuro. Viceversa, ci sono casi in cui il passato non è un buon predittore del futuro e in questo caso i modelli predittivi non sempre funzionano bene. In genere le previsioni si ottengono analizzando le serie temporali ricavate dei dati (log).

C'è da sottolineare che qualsiasi attività di monitoraggio della rete deve rispettare le normative in vigore nello stato in cui si trova il sistema informatico e le politiche interne aziendali. In caso contrario esiste il rischio di essere

perseguiti penalmente per violazione della privacy degli utenti della rete e per intercettazione abusiva di comunicazione informatica. Non sarà, inoltre, possibile utilizzare i dati raccolti per intraprendere azioni legali nei confronti di un intruso.
In questo contesto il DSMS deve essere in grado di memorizzare e analizzare un grosso flusso di dati in arrivo dai log e deve essere in grado di rilevare in real-time eventuali anomalie o intrusioni, scovare pattern comuni nei comportamenti dei clienti ed eseguire previsioni future. Questo richiede di incrociare i dati provenienti da più stream, eseguire dei filtri complessi (clausola 'WHERE' complicata) ed effettuare analisi statistiche avanzate. Tutti questi requisiti richiedono sia l'analisi 'online' dei dati 'recenti', sia una approfondita analisi dei dati storici per ricavare trend ed informazioni statistiche. Le query che possono essere richieste al sistema sono del tipo:

- Trova tutte le pagine web di un particolare server che sono state accedute negli ultimi 50 minuti da più di 200 utenti.
- Esamina tutti i log dei web server in real-time e reindirizza gli utenti nel backup server se il server primario è sovraccarico.
- Estrai da un set di telefonate la chiamata che ha attraversato il maggior numero di stazioni base distinte.

2.3 Panoramica sui sistemi attuali

In questa sezione verranno presentati i principali Data Stream Management System e le loro principali caratteristiche e contesti di applicazione.

2.3.1 Aurora

Aurora è un sistema di gestione di dati a flusso nato da un progetto di collaborazione fra la Brandeis University, la Brown University ed il Massachussets Institute of Technology [1].
Il suo compito è gestire lo stream di dati in arrivo da una varietà di sorgenti ad intervalli regolari o irregolari. Ciascuna sorgente ha un suo identificativo univoco

e Aurora gestisce il timestamp di ciascuna tupla entrante. L'architettura implementata dal sistema Aurora è basata sul paradigma a 'scatole' e 'frecce' tipico dei modelli *workflow* e una sua rappresentazione è mostrata in figura 2.6. La

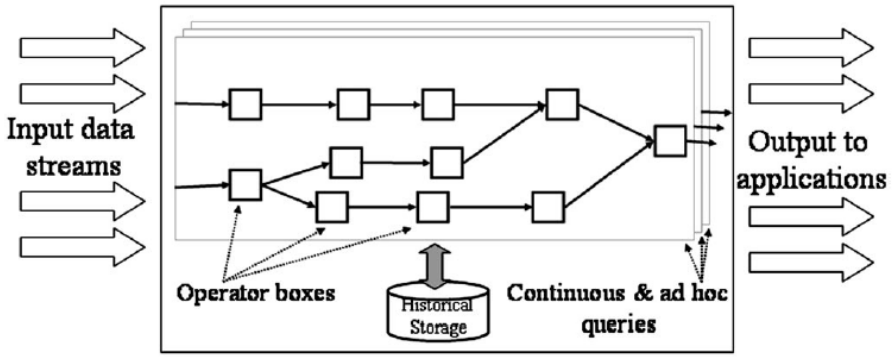
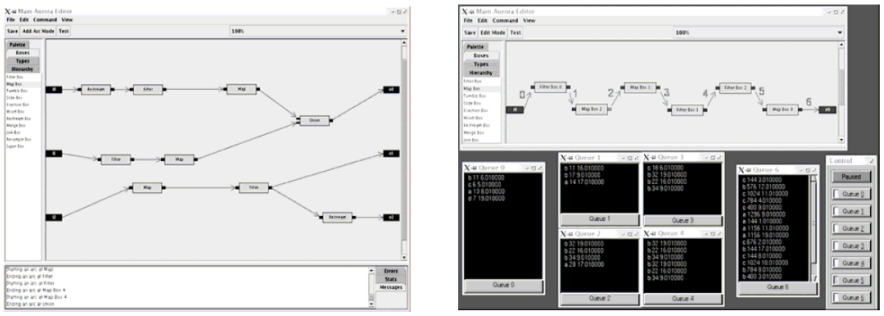


Figura 2.6: Architettura del sistema AURORA.

figura mostra come il sistema si frapponga fra un set di sorgenti di dati a flusso e un insieme di applicazioni. Esso mette a disposizione delle query un insieme di operatori (*operator boxes*) che sono capaci di operare su dati a flusso e che possono essere concatenati per ottenere il risultato atteso dalle applicazioni in fondo alla catena come mostrato in figura 2.8. Ogni insieme di operatori in cascata rappresenta una query ad hoc o di tipo continuo (continuous query); inoltre le query possono essere progettate per salvare una versione compressa oppure una certa porzione dei dati a flusso che stanno analizzando all'interno del repository dei dati storici (*Historical Storage*) e tali dati rimarranno a disposizione delle applicazioni grazie alle query ad-hoc. Il grafo individuato dagli operatori in Aurora è aciclico (vedi figura 2.7.a) ed inoltre ogni grafo di esecuzione implementa delle code che fungono da buffer per le tuple in arrivo dalle sorgenti o dagli operatori a monte (vedi figura 2.7.b).

È importante sottolineare come il sistema assuma che i dati provengono da una serie di sorgenti i cui dati, a differenza dei sistemi transazionali tradizionali, non hanno origine da transazioni generate da attività umane. In questa visione il sistema deve essere di supporto all'operatore affinchè, tramite un certo numero di query sui dati a flusso, sia possibile attivare allarmi o notificare il verificarsi di particolari eventi e condizioni analizzando continuamente i dati che stanno affluendo dalle sorgenti. Il sistema Aurora non fa differenza tra generatori di dati



(*a*) (*b*)

Figura 2.7: GUI del sistema Aurora con un grafo di esecuzione: (a) a sinistra si notano gli stream sorgenti, al centro il grafo composto dai vari operatori ed a destra gli stream di uscita disponibili per le applicazioni client; (b) la fase di monitoraggio delle code implementate da un particolare grafo di esecuzione.

software o hardware (sensori), entrambi sono definiti *data source*. La collezione dei dati generati da un particolare data source viene indicata col termine *stream*. Il sistema Aurora fornisce agli utenti un insieme di operatori primitivi predefiniti, appositamente progettati per operare sugli stream di dati continui. I tipi di query supportate dal sistema sono:

- **Query continue**: Possono operare su un insieme di stream di input e su una certa finestra temporale oppure, in alternativa, su di un certo insieme di tuple per volta.

- **Query Ad-Hoc**: Sono query supportate per permettere alle applicazioni di fare interrogazione sui dati storici memorizzati nell'apposito repository del sistema.

- **Viste**: Permettono di riorganizzare e di filtrare i dati in appositi schemi richiesti dalle applicazioni.

Una caratteristica particolarmente rilevante del sistema Aurora è che, ad ogni query, viene associato un grafo di Qualità del Servizio (QoS, Quality of Service). Aurora tenta sempre di massimizzare il valore di QoS percepito per gli output che produce (i risultati delle query), che sono a loro volta degli stream. Il QoS è una funzione multidimensionale di una serie di attributi del sistema Aurora:

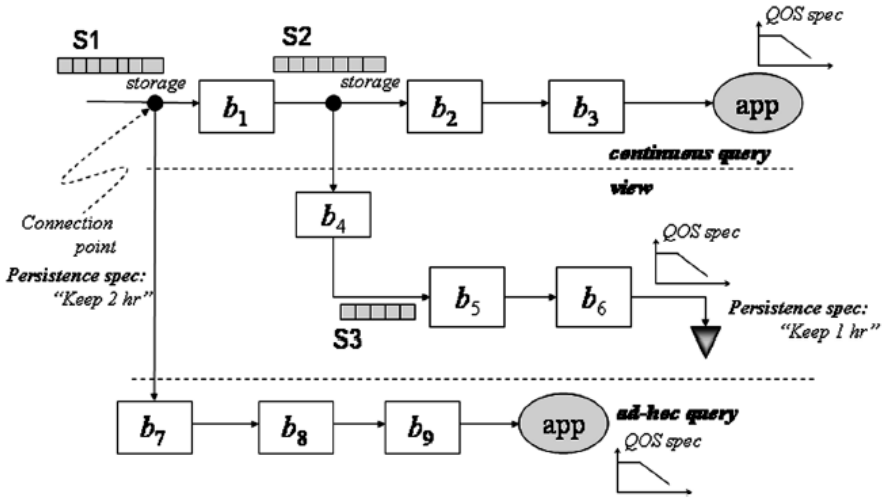


Figura 2.8: Aurora query model.

- *Tempi di risposta*: Le tuple in uscita vengono fornite tenendo conto dei tempi di risposta prestabiliti, altrimenti il QoS degrada proporzionalmente all'aumentare dei tempi di attesa.

- *Eliminazione delle tuple*: Se alcune tuple vengono eliminate per abbassare il carico del sistema, il QoS percepito in output degraderà.

- *Valori prodotti*: il QoS dipende da quanto i valori prodotti in output sono rilevanti e riconducibili al flusso di partenza.

Chiedere all'amministratore di ogni applicazione di fornire una descrizione della funzione multidimensionale QoS desiderata e' ovviamente improponibile. L'approccio scelto dai progettisti di Aurora è molto piu ragionevole. Un amministratore deve specificare, per ogni stream al quale e interessato solo il grafico che lega il QoS al ritardo dei dati ricevuti (vedi figura 2.9). Dal grafico si puo notare come il QoS sia massimizzato se il ritardo in uscita e minore del valore *delta* δ specificato dall'amministratore. Gli altri due grafici mostrati in figura sono opzionali e permettono rispettivamente di: (i) specificare la percentuale di tuple consegnate in uscita (rispetto a quelle arrivate in ingresso) e (ii) indicare al sistema l'importanza di alcuni valori rispetto ad altri. Di particolare rilevanza sono gli operatori a finestra (windowed operators), che permettono di operare su più tuple di dati contemporaneamente, essendo fissata la finestra temporale di

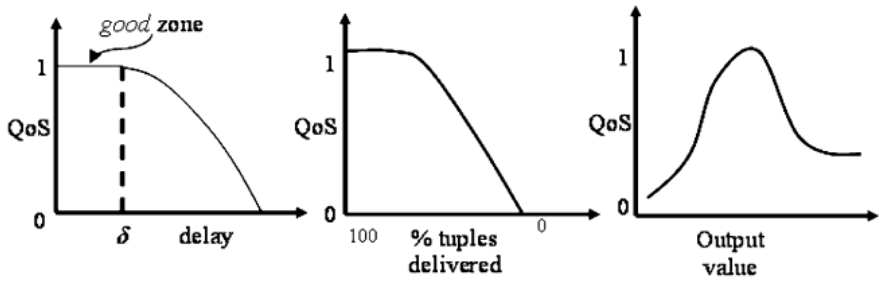


Figura 2.9: Grafici relativi al QoS.

interesse. Ogni operatore a finestra può applicare una funzione di input definita dall'utente, per poi avanzare alla finestra temporale successiva e catturare le nuove tuple da rendere disponibili al prossimo ciclo di esecuzione. Questa funzionalità di spostamento fra le finestre è offerta in modo differente a seconda dell'operatore scelto:

- *Slide*: Permette di spostare la finestra di n tuple in avanti.
- *Tumble*: Garantisce che due finestre consecutive non abbiano mai tuple in comune.
- *Latch*: Assomiglia all'operatore *Tumble* ma è capace di mantenere uno stato fra una operazione e l'altra. E' utile quando si deve calcolare, ad esempio, un valore massimo fra tutte le finestre.
- *Resample*: Produce uno stream parzialmente sintetico interpolando le tuple di uno stream di input e fornendo un nuovo stream in uscita.

Oltre agli operatori a finestra in Aurora sono disponibili altri operatori che permettono di manipolare una tupla per volta:

- *Filter*: Controlla le tuple in uno stream e seleziona solo quelle che verificano una particolare condizione (come la clausola 'WHERE' dell'SQL).
- *Drop*: E' un caso speciale di *Filter* che elimina delle tuple in modo random con una frequenza specificata da un operatore di input.
- *Map*: Permette di applicare una funzione ad ogni tupla dello stream.

- *GroupBy*: Partiziona le tuple di vari stream in un nuovo stream di uscita contenente gli stessi valori rispetto ad un set di attributi di input.

- *Join*: Accoppia le tuple provenienti da più stream di input la cui distanza (definibile dall'utente) ricade in un certo intervallo.

2.3.2 TelegraphCQ

Il sistema TelegraphCQ è nato all'inizio del 2000 ed è il risultato di una attività di ricerca ed implementazione della Berkeley University [22]. Il sistema TelegraphCQ viene definito dai suoi progettisti come un sistema per l'esecuzione di query continuamente adattative su un dataflow ad utilizzo delle applicazioni finali. L'interesse é sempre la gestione di un grande numero di continuous query che operano su un numero altrettanto grande di data streams. Un aspetto interessante da prendere in considerazione è che, nonostante il gruppo della Berkeley abbia ampio accesso al codice sorgente di BerkeleyDB, abbia scelto un altro DBMS Open Source, PostgreSQL, da cui partire per implementare TelegraphCQ.

I progettisti di TelegraphCQ fanno notare un aspetto fondamentale che differenzia i sistemi DBMS tradizionali da quelli a stream. Nei sistemi tradizionali, infatti, è possibile operare l'orchestrazione dei dati e l'ottimizzazione delle query perché è il sistema stesso che con operazioni di *pull* preleva i dati, ad esempio, da un disco. Nei sistemi a flusso, invece, i dati arrivano continuamente al sistema con un pattern di tipo push, il che cambia drasticamente sia le modalità di gestione e memorizzazione dei dati che le modalità di calcolo delle query. Una delle prime versioni del sistema TelegraphCQ ad essere stata implementata, aveva un'architettura ottimizzata per supportare dati web fortemente annidati. Era sostanzialmente un sistema che non poneva l'accento sul supporto efficiente di query concorrenti quanto piuttosto sul supporto fortemente interattivo di query formulate da un singolo operatore. Una delle estensioni proposte dalla Berkeley University per Telegraph é CACQ. Il sistema CACG è un modulo sperimentale sviluppato alla Berkeley University e serve come dimostrazione che i concetti ideati dal gruppo di Berkeley relativamente all'esecuzione di query continue su dati a flusso fossero effettivamente realizzabili. I componenti di cui è composto il sistema sono:

- *Eddie*: Componente specializzato in complesse attività di routing adattativi delle tuple provenienti dagli stream.

- *SteM*: Dedicato a bufferizzare, indicizzare, fare join ed eventualmente eliminare le tuple instradate da componenti Eddie.

Il sistema Telegraph, pur supportando l'esecuzione contemporanea di piú query su un dato numero di stream, non prende in considerazione i seguenti importanti parametri per un sistema DSMS (Data Stream Management System):

- Processa soltanto i dati che possono essere contenuti in memoria centrale.

- Non analizza le prolematiche di scheduling distribuito e gestione delle risorse per query con piccole condivisioni di dati o addirittura senza condivisioni di dati.

- Non utilizza il concetto di QoS per permettere al sistema di adattarsi ad ambienti con risorse limitate.

- Non considera la possibilità di variare il grado di adattività del sistema per reagire ai sovraccarichi e migliorare la flessibilità.

Dopo avere fatto queste esperienze il gruppo della Berkeley ha completamente riprogettato il sistema Telegraph per ottenere TelegraphCQ. L'architettura del sistema presenta una metafora a moduli, ognuno dei quali appartiene ad una certa classe e risolve un particolare problema.
I moduli si dividono sostanzialmente in tre grandi categorie:

- *Ingresso e Caching*: Questa tipologia di moduli sono responsabili dell'interfacciamento con i data source esterni. Molti dei moduli preesistenti sono dei tradizionali wrapper verso l'HTML, l'XML oppure proxy verso le reti peer-to-peer. Altri sono moduli per leggere dati contenuti nel file system, altri ancora per utilizzare dati memorizzati in data base locali o remoti. Alcune implementazioni di moduli possono implementare una cache di dati per evitare i delay introdotti dalla rete.

- *Esecuzione di Query*: Nel sistema TelegraphCQ le query sono eseguite facendo arrivare le tuple ai moduli di query. Questi moduli sono messi in cascata e sono versioni non-bloccanti dei classici operatori relazionali quali:

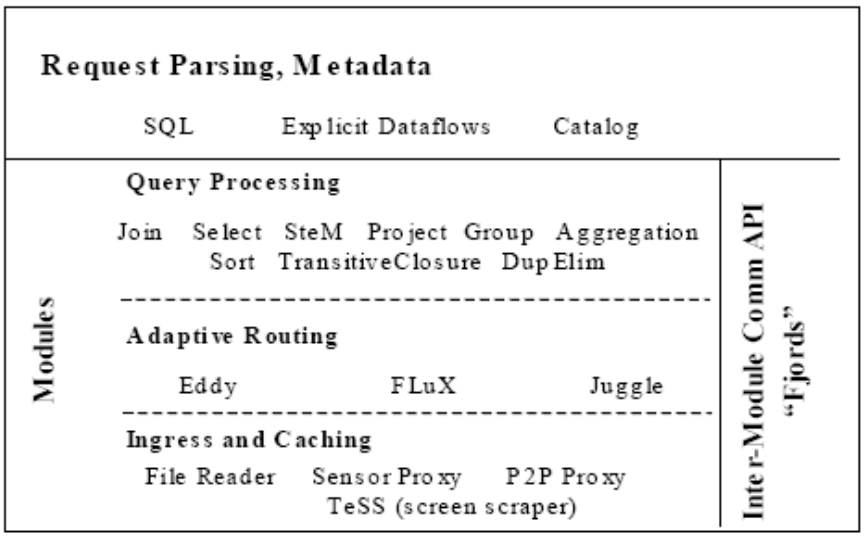


Figura 2.10: Architettura TelegraphCQ.

- Join.
- Selezione.
- Proiezione.
- Raggruppamento ed aggregazione.
- Eliminazione dei duplicati (Distinct).

TelegraphCQ utilizza anche operatori chiamati **SteM** (State Module, modulo a stato). Uno SteM é un repository temporaneo di tuple. É capace di memorizzare tuple generate durante l'esecuzione di query e supporta le seguenti operazioni:

- Inserimento (*build*).
- Ricerca (*probe*).
- Cancellazione (*Eviction*).

- *Routing Adattativo*: Il sistema TelegraphCQ non si basa sui query plan tradizionali, che sono essenzialmente statici. Costruisce, invece, un query plan che contiene moduli di routing adattativo, che sono capaci di ottimizzare il piano della query in modo continuo durante l'esecuzione. I moduli di tipo **Eddie** decidono in modo adattativo come instradare i dati ad altri

operatori prendendo decisioni tupla-per-tupla. I moduli di tipo **Juggle** implementano operazioni di ordinamento al volo dei dati (tuple) in base al timestamp oppure al contenuto. I moduli di tipo **Flux** instradano le tuple fra le macchine di un cluster per supportare il parallelismo, il bilanciamento del carico e la tolleranza ai guasti.

Per quanto riguarda la semantica delle query, il sistema supporta countinuous queries calcolate su insiemi di tabelle e data streams. Per riuscire a gestire flussi di dati illimitati, alcuni operatori possono essere eseguiti soltanto su finestre di dati di dimensione finita. Ogni finestra é caratterizzata da una posizione iniziale ed una finale che servono a specificare quali tuple sono presenti nella finestra corrente. Al fine di supportare un gran numero di applicazioni, TelegraphCQ mette a disposizione dell'operatore un complesso sistema di gestione delle finestre capace di operare sia su dati che sono giá arrivati che su quelli che ancora devono arrivare dallo stream:

- *Landmark queries*: Il puntatore di coda rimane fisso mentre quello di punta si sposta in avanti all'arrivo delle nuove tuple dallo stream.
- *Sliding queries*: Le query a scorrimento fanno spostare contemporaneamente i limiti della finestra appena sono disponibili nuovi dati dallo stream.
- *Loop WindowIs*: Questo particolare costrutto permette agli operatori di personalizzare la gestione delle finestre all'interno delle query. Si usa insieme al costrutto for-loop.

In particolare, una funzionalità interessante di TelegraphCQ è la possibilità di gestire contemporaneamente più notazioni temporali, sia logiche (ad esempio l'ordine di arrivo delle tuple) che fisiche (ad esempio il timestamp). Il tempo è visto come un ordinamento non totale ma parziale ed il sistema offre primitive per passare da un sistema temporale all'altro tramite un'apposita algebra che estende gli operatori relazionali standard.

2.3.3 STREAM

STREAM, STanford stREam datAManager, è un prototipo sviluppato alla Stanford University che implementa un sistema per la gestione di flussi di dati

(DSMS) tramite un architettura client/server. Il client sottomette le query al server e ne riceve i risultati. Il server si occupa di ricevere i data stream ed elaborare le query sottomesse dal client.
All'inizio dell'interazione con l'utente, il sistema permette di eseguire un applicativo client con tecnologia Java Web Start presso la macchina dell'utente (vedi fig. 1). Tale client permette di interagire con il server fornendo all'utente le

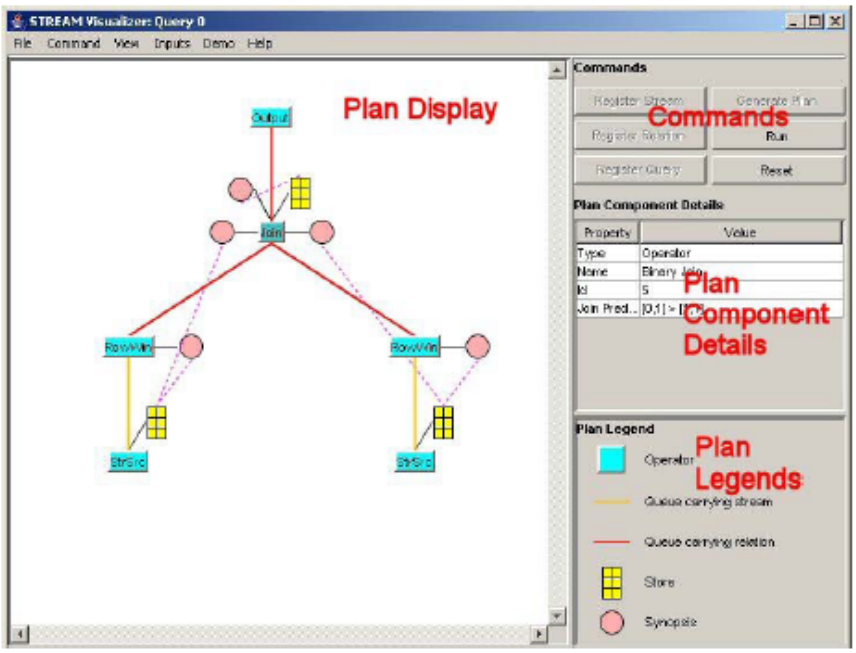


Figura 2.11: Interfaccia STREAM.

seguenti funzionalità:

- Registrare nuovi stream e le loro relazioni presso il server. Questa funzionalità permette di definire, per ogni schema creato dall'utente, lo schema dei dati con le medesime modalità di un DBMS relazionale.

- Registrare le interrogazioni continue presso il server. Le interrogazioni continue verranno invocate periodicamente (con una frequenza scelta dall'utente) ed il loro output potrà essere visualizzato direttamente nel client oppure utilizzato per costruire altre query.

- Vedere i piani delle query (*query plan*) che sono stati calcolati dal server. i piani delle query sono visualizzati tramite un grafo di esecuzione che

mette in evidenza le sorgenti di dati, le join effettuate e le destinazioni dei risultati.

- Eseguire e sospendere l'esecuzione delle query registrate presso il server e vederne i risultati.
- Monitorare le performance del server, delle query e del client. Ad esempio l'utente può monitorare il flusso di tuple al secondo che fluiscono attraverso un certo ramo del grafo che rappresenta un piano di esecuzione (vedi figura 2.12), oppure monitorare l'uso della memoria o del carico della CPU.
- Salvare le esecuzioni, le query o i risultati sul file system locale del client.

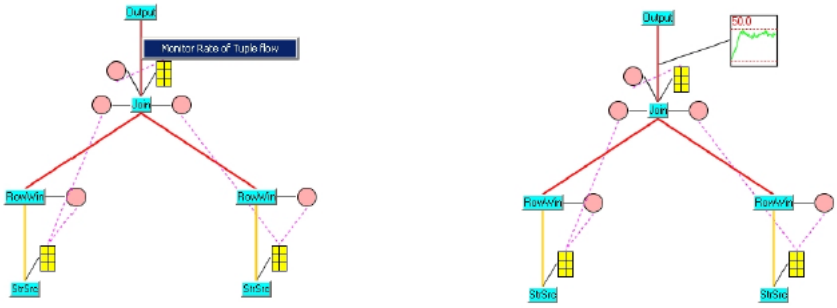


Figura 2.12: Definizione di monitor di attività su un elemento del grafo di esecuzione di una query. A sinistra viene mostrata la selezione del collegamento da monitorare. A destra, viene creato un monitor che riporta in un diagramma temporale i valori che attraversano il collegamento.

Al fine di mostrare come il sistema STREAM funzioni, possiamo mostrare un esempio. Definiamo, ad esempio, due flussi (stream) aventi schema simile:

```
S1 (name char 20, value int)
S2 (name char 20, value int)
```

A questo punto definiamo nel sistema una continuous query che permetta di fare una join fra i due flussi:

```
SELECT *
FROM S1 [rows 20], S2 [rows 20]
WHERE S1.name = S2.name
```

```
AND S1.value > S2.value;
```

Una volta che la query è stata registrata presso il server, il server calcola un query plan e abilita il client ad eseguirla. Dopo aver avviato l'esecuzione, l'utente può tenere sotto controllo le attività creando una serie di monitor, ognuno collegato all'entità che si desidera tenere sotto controllo (vedi figura 2.12).
Le tipologie di monitor che è possibile attivare sugli elementi di un query plan sono:

- Controllo della frequenza di arrivo delle tuple: Il monitor mostra un grafico del tempo che si adatta ai valori rilevati e che mostra un numero in alto a sinistra indicante il valore massimo dell'asse y.
- Uso della CPU: mostra il carico della cpu rispetto alle query attivate dall'utente.
- Uso della memoria: mostra lo stato di uso della memoria.

Il linguaggio usato per esprimere le query in STREAM è chiamato CQL (Continuous Query Language) ed è una estensione del linguaggio SQL.

2.3.4 NiagaraCQ

Il sistema Niagara può essere utilizzato per la gestione, il retrieval oppure per monitorare cambiamenti interessanti nel tempo di dati di tipo XML. Queste funzionalità sono implementate dai suoi tre principali componenti: il *Search Engine*, il *Query Engine* ed il *Trigger Manager*. Data la crescente popolarità che il linguaggio XML sta raggiungendo nella rappresentazione di dati semi-strutturati sul web e come linguaggio standard per lo scambio di dati, la rilevanza del sistema Niagara è abbastanza evidente. Inoltre XML è per natura auto-descrittivo e quindi attività di retrieving di dati XML risultano sicuramente semplificate se paragonate a quelle su file di testo piatti. Per esempio, se si ricercano informazioni su una nave di nome 'Stoccolma', tutti i moderni motori di ricerca quasi sicuramente includeranno fra i risultati anche informazioni sulla città di Stoccolma, che non sono, ovviamente, di interesse. Con il sistema Niagara, invece, è possibile ottenere risposte più precise in quanto il motore XML cercherà gli elementi di tipo nave che includono un tag nome con valore 'Stoccolma'.

2.3.5 Cougar

Il sistema Cougar, sviluppato presso la Cornell University, è un esempio di implementazione di DSMS per una Sensor Network (rete di sensori) [26]. L'approccio innovativo adottato dal sistema consiste nel farsi guidare dalle query degli utenti per generare un query plan ottimizzato da eseguire direttamente nella rete di sensori per minimizzare le comunicazioni e massimizzare la durata delle batterie a bordo dei sensori. Le reti di sensori, come visto precedentemente, hanno forte limitazioni in termini di qualità delle comunicazioni, durata della batteria, capacità di elaborazione e disturbo del segnale.
Nel sistema Cougar i progettisti hanno pensato ad un approccio alla rete di sensori come database distribuito. Questa scelta scaturisce principalmente da due considerazioni:

- Le query dichiarative risultano particolarmente adatte per l'interazione con una rete di sensori. Gli utenti e le applicazioni rivolgono le loro interrogazioni senza conoscere i dettagli del sistema, come i dati vengono generati e come vengono manipolati per rispondere alla query.

- Essendo i nodi sensore alimentati a batteria, aumentare la vita media della rete di sensori è uno dei maggiori obiettivi nella progettazione di sensor networks. Inoltre la trasmissione di tutti i dati verso un nodo della rete che calcoli poi i risultati delle query è molto dispendioso per reti di dimensioni non banali perché la comunicazione wireless consuma molta energia. Siccome i nodi hanno una seppur piccola potenza computazionale, parte della computazione delle query può restare nella rete, almeno per quel che riguarda semplici operazioni quali l'aggregazione di tuple oppure l'eliminazione di tuple non rilevanti.

Questo approccio distribuito sposta parte del peso della computazione dal nodo centrale verso i nodi della rete di sensori cercando di minimizzare la quantità di dati che devono essere trasmessi, essendo le operazioni di trasmissione wireless molto più costose di quelle di calcolo per le piccole batterie di cuii sensori sono dotati. Questa architettura particolarmente innovativa contrasta con la maggior parte delle attuali reti di sensori, le quali lavorano principalmente come un mezzo per collezionare i dati ed implementano una rete che permette ai valori rilevati dai sensori di raggiungere un nodo centrale su cui avvengono tutte le operazioni

di calcolo, di persistenza e di risposta alle query dei client. Il sistema Cougar prevede un complesso di primitive che permettono di volta in volta di disattivare le comunicazioni di tutti quei nodi che non sono coinvolti nel calcolo delle query attualmente in esecuzione, aumentando notevolmente la vita media (periodo fra una ricarica e l'altra delle batterie) dei singoli nodi. Ogni nodo sensore è dotato di un query proxy layer che è l'interfaccia verso tutti i sistemi esterni, sia quelli di routing che di esecuzione delle query. Nel gateway della rete, che si occupa di collezionare e di rispondere alle query, è posto un modulo di ottimizzazione delle query (*query optimizer*) che ha il compito, una volta ricevuta una query, di generare per essa un query plan. Il query plan viene creato in accordo con il catalogo dei nodi (una sorta di fotografia della rete di sensori) e con il tipo di calcoli che la query richiede. A questo punto il piano della query viene trasemsso nella rete, ma soltanto verso i nodi interessati. Esso contiene una descrizione dettagliata sia delle comunicazioni fra i sensori che l'esatto piano di computazione che dovrà avvenire su ogni singolo nodo.
Per chiarire tutti gli elementi coinvolti in una query è opportuno fare un esempio. Supponiamo di avere una query continua Q per monitorare la temperatura media in un ufficio ogni t secondi.
La query Q invia una notifica,ovvero genera un record di output, ogni volta che

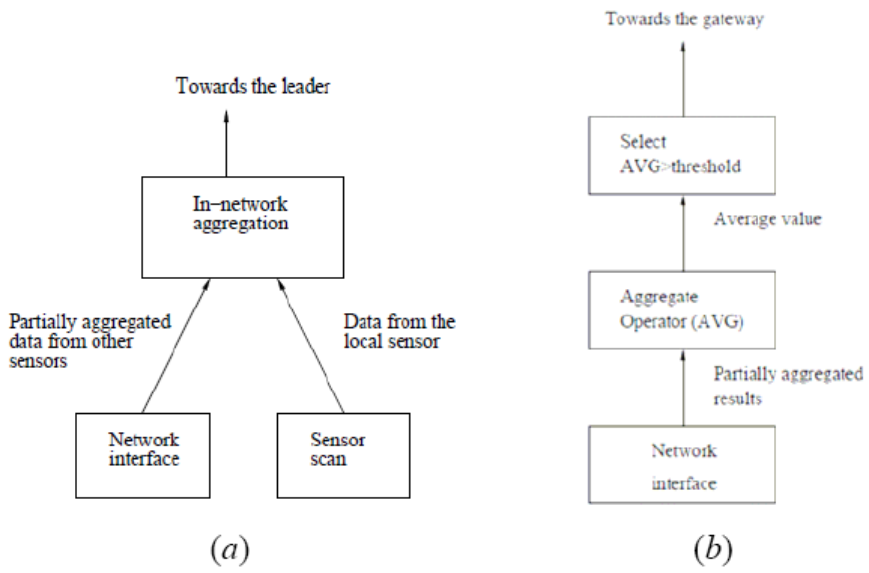


Figura 2.13: Query plan del nodo sorgente e del nodo leader.

la temperatura nell'ufficio è maggiore di un certo valore di soglia specificato dall'utente. Come primo passo per la valutazione di tale query il sistema **gateway** ne genera una versione ottimizzata cercando di vedere se una o più sottoparti di tale query sono in comune con altre già in esecuzione. Assumendo, per semplicità, che la query Q sia l'unica in esecuzione nella rete, il gateway passerà a calcolare un query plan QP. Il query plan QP contiene le informazioni per individuare il **leader** della query, cioè il nodo eletto come nodo di computazione per la query presso il quale i calcoli necessari (temperatura media in t secondi) saranno eseguiti. Il nodo leader potrebbe essere un nodo che ha ancora un buon livello di carica della batteria oppure un nodo casuale scelto dall'algoritmo distribuito di elezione del leader. I piani di computazione prodotti dal gateway saranno due, uno per il nodo leader ed uno per i rimanenti nodi della regione di query. In figura 2.13.a si può vedere il query plan di un nodo non-leader. In questi tipi di nodo c'è un modulo *scan* che acquisisce periodicamente i valori e li invia verso il nodo leader. Inoltre il query plan contiene anche un operatore di aggregazione per i dati ricevuti dagli altri sensori. In figura 2.13.b vediamo invece il query plan del nodo leader contenente l'operatore AVG (media) che serve per calcolare la media delle temperature ed un operatore SELECT che serve per controllare se abbiamo superato il livello definito dall'utente.

2.3.6 Linear Road

Linear Road è un benchmark per i sistemi DSMS creato dai progettisti di Aurora e STREAM [5].
E' stato creato per misurare quanto un sistema DSMS è performante in un contesto dove si verifica un grosso flusso di dati ed è indipendente da come il DSMS è stato creato. Richiede di analizzare e gestire sia i dati recenti che quelli storici. In particolare, Linear Road simula il traffico urbano in cui il calcolo del pedaggio su ogni strada e per ogni veicolo è stimato dinamicamente sulla base di alcuni fattori, come le congestioni o gli incidenti nelle vicinanze. Il pedaggio aumenta se c'è molto traffico su quella strada in modo da scoraggiare gli autisti dall'utilizzare quella strada evitando quindi di alimentare la congestione.
Linear Road rispetta le seguenti caratteristiche:

- I dati generati sono semanticamente corretti, ovvero i report delle autovetture non sono generati a random ma sono aderenti con la realtà. Ad

esempio, se la posizione della vettura viene riportata ogni 10 secondi, non è possibile che tra due report successivi la vettura si sia spostata oltre un certo limite (ad esempio 100km di distanza tra 1 report e il successivo). Questo caratteristica sottolinea il fatto che Linear Road non è soltanto un generatore di dati ma un vero e proprio simulatore di traffico urbano.

- Le metriche utilizzate per valutare le performance delle query continue sono essenzialmente due:
  - **Tempo di risposta**: Viene calcolato come la differenza in valore assoluto tra il tempo di arrivo al tempo di uscita della tupla dal DSMS.
  - **Carico di lavoro supportato**: Si controlla che i requisiti prestazionali vengano rispettati. Se il tempo eccede la soglia prevista, significa che il sistema non riesce a supportare un certo carico di lavoro.
- Non è previsto nessuno standard riguardo al linguaggio di esecuzione delle query continue sullo stream di dati, pertanto il benchmark è indipendente dal linguaggio di interrogazione usato.
- Viene considerata la possibilità di avere più risposte corrette per una stessa query in base all'ordine di arrivo delle tuple o allo stato precedente di alcuni elementi.

I dati che Linear Road invia al DSMS sono i *report sulla posizione* dei veicoli (ogni 30 secondi) ed eventualmente delle *query storiche* che vengono emesse con una certa probabilità ogni volta che si emette un report sulla posizione.
Il DSMS deve:

- Mantenere le statistiche sul numero medio di veicoli e velocità media per ogni strada. Queste informazioni devono essere aggiornate almeno ogni minuto.
- Calcolare il pedaggio per ogni strada sulla base del traffico su di essa e alla presenza di incidenti nelle vicinanze. In genere il DSMS mantiene una finestra temporale di 5 minuti sui report per quella strada e ne fa una statistica per calcolarne il pedaggio.

- Verificare gli incidenti e comunicarli alle vetture nelle vicinanze. L'evento viene tenuto in memoria fintanto che non viene risolto.

- Poter ricevere delle query da parte degli automobilisti che possono richiedere: il bilancio completo dei suoi pagamenti (*account balance*), il totale dei pedaggi pagati su una certa expressway in un certo giorno (*daily expenditure*) e il tempo di attraversamento di una certa expressway in un certo giorno (*travel time*). Queste query richiedono una gestione storica delle informazioni.

Ciascuna risposta alle query deve essere fornita correttamente entro certi tempi con un certo carico di lavoro. Il punteggio finale che viene assegnato al DSMS è chiamato *L-Rating*, dove L indica il numero di strade (*expressway*) che il sistema riesce a gestire, rispettando i vincoli temporali e di correttezza imposti. L-Rating misura la scalabilità e l'efficienza del DSMS.

Una 'Linear City' e' composta da diverse *expressway*, le quali sono a loro volta composte da 8 corsie, 4 che procedono in una direzione e 4 che procedono nella direzione opposta, come mostrato in figura 2.14.

La grandezza media di una linear city è 100x100 miglia, composta da 10 expressway posizionate in orizzontale e distanti 10 miglia l'una dall'altra.
Per 'segmento' si intende un tratto di expressway lunga 1 miglio. Il simulatore genera un certo numero di veicoli che eseguono un certo viaggio. Il viaggio inizia su una certa rampa di un certo segmento e finisce sulla rampa di uscita di un'altro segmento della stessa expressway.
Ciascuna vettura genera un report sulla posizione ogni 30 secondi con un certo timestamp che indica i secondi trascorsi da quando è stato fatta partire la simulazione. La tupla ha questa struttura:

```
(Type = 0, Time, VID, Spd, XWay, Lane, Dir, Seg, Pos)
```

Type = 0 indica che la tupla indica il report di una posizione. `Time`$(0...10799)^3$ è il timestamp. VID$(0...MAXINT)$ è l'identificativo della vettura. `Spd`$(0...100)$ indica la velocità della vettura in miglia orarie. `XWay`, `Lane`, `Dir` e `Seg` sono funzioni delle coordinate (x, y) riportate dalla vettura:

- `XWay` $(0...L-1)$: Indica la expressway sulla quale il veicolo si trova.

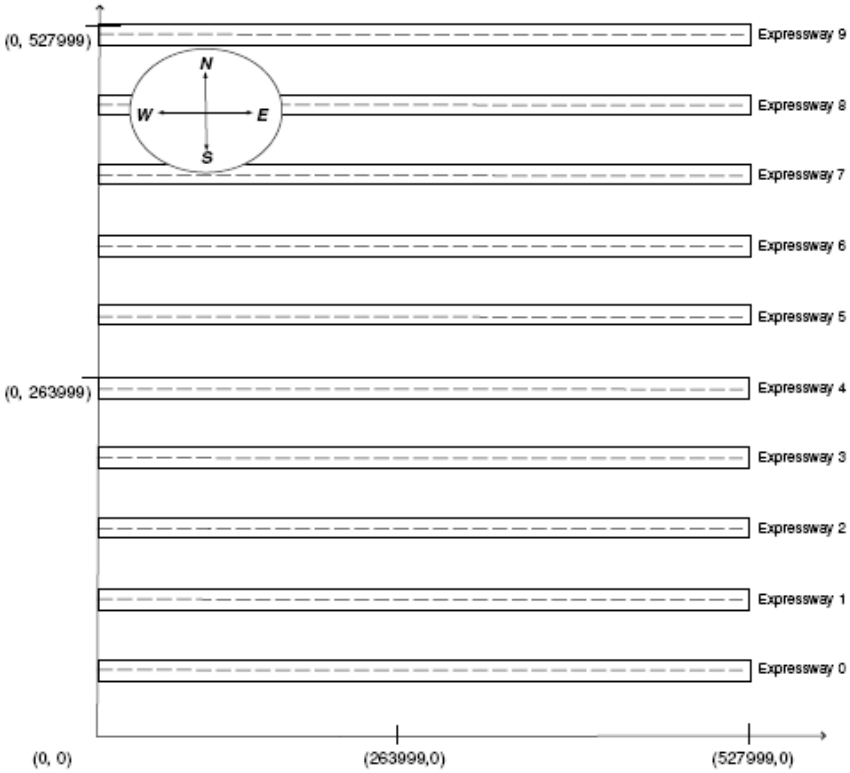


Figura 2.14: Linear City.

- `Lane` (0...4): Indica la corsia sulla quale il veicolo si trova. La corsia 0 è di entrata mentre la 4 è quella di uscita.

- `Dir` (0...1): Indica la direzione di percorrenza del veicolo. 0 indica che il veicolo stà procedendo verso est, 1 verso ovest.

- `Seg` (0...99): Indica il segmento di percorrenza dei veicolo.

Gli incidenti vengono generati a random su una certa corsia di una expressway ogni 20 minuti. La rilevazione di un incidente avviene quando 2 veicoli si trovano fermi nello stesso punto per più di 5 report consecutivi. Il tempo medio per risolvere un incidente è compreso tra i 10 e i 20 minuti e avviene quando le vetture tornano in movimento.
Con l'1% di probabilità un report sulla posizione è accompagnato da una query storica.

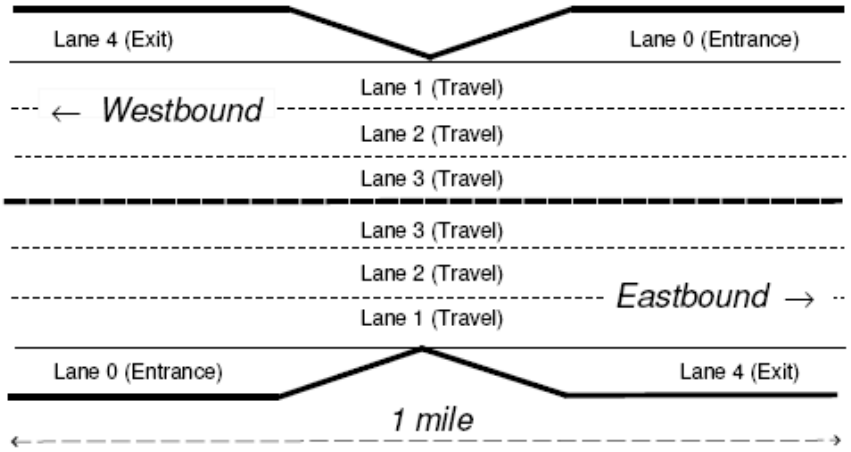


Figura 2.15: Struttura di una expressway.

Linear Road fornisce anche un file contenente i dati storici relativi a 10 settimane di traffico e contiene i pedaggi per ogni corsia delle expressway giorno per giorno.
Il DSMS deve calcolare il pedaggio ogni volta che il veicolo si sposta da un segmento ad un altro. Il pedaggio calcolato verrà notificato al veicolo e aggiunto al bilancio (dato storico). Se il veicolo esce dal segmento sulla rampa di uscita, allora non gli verrà notificato il pedaggio.
Il sistema Linear Road è un sistema composto da 4 diversi tool:

- **Generatore di dati storici**: Genera i file che contengono la storia passata sui pedaggi delle ultime 10 settimane.
- **Simulatore di traffico**: Genera i file che contengono i dati da dare in input al benchmark.
- **Data driver**: Strumento che legge i file creati dal simulatore e li spedisce al DSMS come se fossero in real-time.
- **Validatore**: Verifica la correttezza delle risposte fornite dal DSMS e ne misura le performance.

Parte II

Gestione efficiente dei dati in un DSMS

Capitolo 3

Architettura di un DSMS efficiente e scalabile

Obiettivo di questo capitolo è descrivere in dettaglio l'architettura di un innovativo Data Stream Management System in cui il modulo di store manager è separato rispetto al modulo di query engine [7]. Verrà descritta l'architettura di funzionamento, le strutture dati utilizzate, i modelli di accesso, le regole per l'accesso ai dati e le caratteristiche di un sistema di querying efficiente e scalabile.

3.1 Descrizione generale dell'architettura

Le moderne applicazioni di stream processing prevedono l'esecuzione parallela di numerose query continue, la gestione online (real-time) di dati recenti e offline dei dati storici e richiedono una gestione dinamica delle finestre e degli operatori. Esempi di queste applicazioni si possono ricercare nel monitoraggio del traffico urbano, nell'analisi del traffico internet, nel data processing di informazioni scientifiche, nell'analisi dei log e molte altre applicazioni.

La maggior parte delle soluzioni DSMS presenti sul mercato o in via sperimentale non offrono un approccio chiaro e sistematico circa la gestione dello storage. In questi sistemi, lo storage manager è fortemente accoppiato con il Query Processing Engine. Un design di questo tipo non solo rende lo storage manager ad-hoc e poco flessibile, ma limita fortemente le possiblità di ottimizzazioni per ciascun contesto applicativo nel quale il sistema viene inserito.

Risulta invece evidente che avere uno storage manager in grado di adattarsi e ottimizzarsi per ciascuna applicazione è la chiave per avere buone performance.

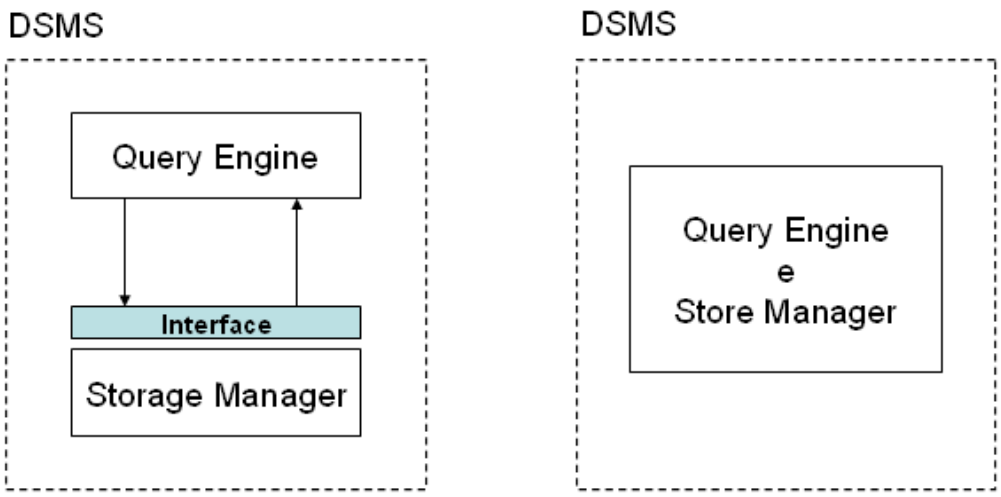


Figura 3.1: A sinistra, un DSMS in cui la componente di gestione della memoria è separata rispetto alla gestione delle query. A destra, la struttura dei tradizionali DSMS.

Nei DBMS tradizionali il motore di query processing è separato rispetto allo storage manager e questo risulta fondamentale per poter ottenere molti ottimizzazioni in grado di migliorare notevolmente le performance.
L'architettura studiata e implementata prevede quindi la creazione di un DSMS in cui la componente di gestione della memoria e quella di gestione delle query sono ben separate e distinte, in modo da poter poi ottimizzare lo storage manager sulla base del data stream in arrivo e delle query in esecuzione. Saranno quindi previsti una serie di parametri in grado di modificare la configurazione dello storage manager migliorandone notevolmente le performance. In particolare, lo storage manager sarà in grado di adattare dinamicamente le strutture dati in base al data stream in arrivo e alle query sottomesse dal Query Processing Engine tramite una opportuna interfaccia di comunicazione. Tale interfaccia permetterà al Query Processing Engine di:

- Sottomettere query continue o ad-hoc e ricevere i risultati.
- Creare nuovi *storage point*.

- Configurare alcuni parametri dello storage manager.

Da questo punto di vista lo store manager è molto simile al gestore del buffer dei normali DBMS. I DBMS devono gestire una grossa mole di dati, e nel corso delle elaborazioni lo spazio richiesto per i blocchi di dati sarà spesso maggiore dello spazio di memoria disponibile. Per questo vi è la necessità di gestire un'area di memoria in cui caricare e scaricare i blocchi. Il gestore del buffer si occupa principalmente di gestire le operazioni inerenti il salvataggio e il caricamento dei blocchi da memoria principale a memoria secondaria. Similmente, lo storage manager della nostra architettura dovrà gestire lo stream di dati in arrivo e migrare su disco i dati 'consumati' o quelli troppo vecchi per le operazioni real-time. Ci sono però due sostanziali differenze:

- Bisognerà gestire stream di dati anzichè dati statici. Conoscere come le informazioni entrano nel sistema (*update pattern*) è fondamentale per assicurare una acquisizione dati efficiente e performante.
- Nei DSMS spesso le query sono conosciute a priori (*read pattern*) e questa conoscenza permette di eseguire molte ottimizzazioni riguardo alle strutture dati da usare, le modalità di accesso e gli indici.

Nelle prossime sezioni verranno analizzati meglio i parametri che influenzano le performance e la scalabilità del sistema.

3.2 Store Management System (SMS)

In questa sezione verranno descritti i principali parametri di configurazione di cui uno store manager deve tenere conto per potersi adattare in maniera dinamica ed efficiente a ciascun contesto applicativo. Si descriveranno inoltre le strutture dati e gli indici utilizzati per gestire lo stream di dati.

3.2.1 Role e Access Model

In un DSMS uno *store* (o *storage point*) è un area di memoria in cui i dati (tipicamente tuple) provenienti da uno stream vengono memorizzate. Dati i vincoli stringenti di performance richiesti dalle applicazioni che necessitano di

un DSMS, tale area viene mantenuta in memoria principale in una particolare struttura dati.

Ci possono essere due modi con cui i dati passano dallo storage manager al query engine: *push-based* e *pull-based*.
Nel modello push-based i dati vengono 'spinti' automaticamente dentro al query network dove vengono filtrati e selezionati dagli operatori presenti. Il modello pull-based è invece molto simile ai tradizionali DBMS dove il dato è 'passivo' e viene prelevato in caso di necessità dalle query.
Dati i requisiti di flessibilità richiesti dal nostro DSMS, devono essere previste entrambe le modalità di funzionamento.

I primi due parametri da gestire riguardano il modo con cui i dati entrano nello storage system e come i dati passano da uno store al query engine:

Role: Questo parametro gestisce come i dati vengono memorizzati all'interno dello store. Uno store può essere *attivo* o *passivo*. Se attivo, i dati entrano automaticamente all'interno dello store e quindi lo store preleva i dati dalla sorgente. Se passivo, ci sarà una query che preleva i dati dalla sorgente e li inserisce all'interno dello store.

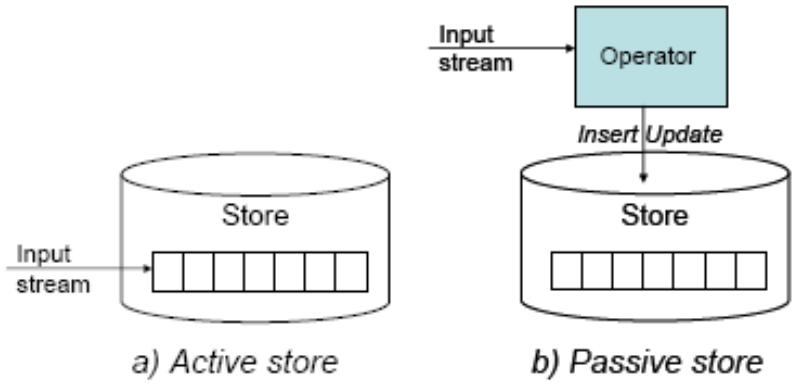


Figura 3.2: Parametro 'Role'.

Access Model: Questo parametro definisce come i dati passano dallo storage manager al query engine. Ci sono due modi: *pull-model* o *push-model*. Il pull-model attende una richiesta da parte di un operatore, il push-model invece notifica automaticamente il query engine che ci sono nuovi dati da elaborare.

Quest'ultima modalità prevede che il query engine abbia a sua volta una interfaccia di comunicazione che permetta allo storage manager di avvisare il query engine della presenza di nuovi dati.

3.2.2 Sincronizzazione, Schema e Persistenza

Il query engine ha uno *scheduler* che decide l'ordine di esecuzione dei job (operatori). Un compito che non riesce ad eseguire è il ritardare l'esecuzione di un certo operatore fin tanto che una certa condizione sui dati non si è verificata. Questa caratteristica viene chiamata *sincronizzazione*. Ad esempio, si vuole ricevere determinate informazioni solo se sono aggiornate entro l'ultimo minuto, altrimenti si aspetta finchè non vengono aggiornate. Nel sistema Aurora questa funzionalità è implementata grazie ad un operatore denominato *Wait-For*.
Nel nostro sistema la sincronizzazione viene implementata direttamente a livello di storage in modo che il query engine dovrà solamente specificata la condizione che dovrà essere rispettata per poter ricevere i dati e lo store glie li invierà in automatico non appena la condizione sarà soddisfatta. Generalmente questa condizione è relativa al valore di un qualche attributo e questo rende necessaria la conoscenza dello *schema* del data stream.

Da questa analisi vengono estratti altri due parametri di fondamentale importanza: la sincronizzazione e lo schema. Lo schema può essere conosciuto a priori oppure no. Conoscerlo permette allo storage manager di effettuare delle scelte di implementazione più avanzate che verranno analizzate meglio in seguito.

Un altro parametro importante è la *persistenza* dei dati. Alcune query possono richiedere di salvare alcuni dati in memoria secondaria perchè non hanno requisiti real-time. Questa scelta influenza il valore di persistenza che può variare tra: Persistence, se il dato viene salvato in memoria secondaria, Transient, se il dato è mantenuto in memoria primaria.

3.2.3 Parametri Performance-Related

Oltre a garantire le funzionalità e il supporto necessario per il design dell'architettura, lo storage manager deve assicurare certe performance. Come descritto precedentemente, le applicazioni data-intensive richiedono bassi tempi di risposta e alta scalabilità. Alcune informazioni possono essere di grande aiuto

allo storage manager per effettuare delle scelte che vanno incontro a questi requisiti. Le modalità di accesso allo store (*access pattern*) e la condivisione della computazione da parte delle query (*sharing*) sono due parametri molto importanti per permettere allo storage manager di implementare specifici meccanismi che velocizzano le operazioni sugli store:

Access Patterns (Read e Update): L'access pattern è il modo con cui un operatore accede ai dati di uno store. Si può accedere per leggere i dati (*read pattern*) o per aggiornarli (*update pattern*). Conoscendo come gli operatori accedono ai dati si possono eseguire alcune ottimizzazioni che migliorano le performance. Si discuteranno meglio questi aspetti nella prossima sezione.

Sharing: Le query continue mostrano una grande similarità tra loro. Questa similarità può essere sfruttata dal query engine per eseguire delle ottimizzazioni. Si possono, ad esempio, eseguire in parallelo diverse query e mettere a disposizione i risultati intermedi che vengono calcolati agli operatori di query differenti. Questo permette di evitare di eseguire gli stessi calcoli sugli stessi dati, guadagnando in performance e diminuendo lo spreco di memoria. Ad esempio, nel caso del benchmark Linear Road, le informazioni relative agli incidenti possono essere condivise tra la query che si occupa di avvisare gli automobilisti nelle vicinanze e quella che calcola il pedaggio.

Nella tabella 3.2.3 vengono riportati i parametri che influenzano la struttura e il comportamento dello storage manager.

Parametro	Descrizione	Valori Possibili	Default
`Role`	Come il dato entra nello store	Attivo, Passivo	Passivo
`Access Model`	Come il dato passa dallo store al query engine	Push, Pull	Pull
`Sync`	Definisce se lo store deve sincronizzarsi oppure no	Falso, Condizione di sincronizzazione	Falso
`Schema`	Conoscenza sullo schema dei dati	Nessuno Schema, Conoscenza sullo schema	Schema definito
`Persistenza`	Dove il dato viene salvato (memoria principale o disco)	Persistente, Transiente	Transiente
`Read Pattern`	Richiesta di lettura da uno store	Sequenziale, Random, Clustered	Random
`Update Pattern`	Richiesta di aggiornamento di uno store	FIFO, RANDOM, IN-PLACE	RANDOM
`Sharing`	Definisce se lo store è condiviso tra più query in lettura	Condiviso, Non Condiviso	Non Condiviso

3.2.4 Update pattern

In un DSMS è presente un flusso continuo di data items che entra nel sistema, viene processato dalle query e produce dei risultati che escono dal sistema. Durante questo processo i dati vengono salvati in maniera permanente o temporanea

in diversi store. Se i dati sono temporanei possono 'scadere' perchè troppo vecchi e quindi venire cancellati o migrati su memoria secondaria.

Gli item che entrano nel sistema hanno un certo set di attributi che coprono un certo schema e il sistema assegna loro un identificativo univoco. Gli inserimenti negli store avvengono sempre in append. Per *update pattern* si intende una operazione di cancellazione o aggiornamento di un valore (*value replacements*).

Uno store può subire una operazione di update in due occasioni:

- Un data item scade e quindi viene rimosso/migrato.
- Il data item viene consumato da una certa query o da un gruppo di query e quindi deve essere rimosso/migrato.

Per capire meglio, consideriamo di avere un operatore Consumatore e un operatore Produttore con uno store tra i due che memorizza i risultati prodotti dal Produttore e li mette a disposizione del Consumatore (figura 3.3). Il Produt-

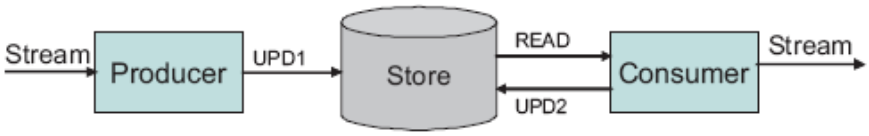


Figura 3.3: Produttore e Consumatore.

tore esegue regolarmente delle operazioni di inserimento di nuovi dati in append utilizzando un particolare update pattern denominato UPD1, mentre il Consumatore ha un suo update pattern (dato dal consumo dei dati) denominato UPD2 e un read pattern denominato READ. UPD1 è il risultato della scadenza dei dati, UPD2 è il risultato del consumo dei dati.

Come si vedrà in seguito, combinando l'update pattern del Produttore con quello del Consumatore si influenza la struttura e il funzionamento dello store. Nella figura 3.4 si mostrano alcune query con le relative modalità di accesso ai dati.

```
SELECT Attr1, Attr2
FROM Stream
```

a. Never Expire

```
SELECT Attr1, Attr2
FROM Stream
WINDOW 24 HOURS
```

b. Ordered Expire

```
ON Stream S
DELETE FROM Store W
WHERE W.Attr1 = S.Attr1
```

c. Unordered Expire

```
ON Stream S
UPDATE Store W
SET W.Attr2 = W.Attr2 + S.Attr2
WHERE W.Attr1 = S.Attr1
```

d. Replaced Expire

```
SELECT *
FROM Store
```

e. Never Consume

```
SELECT Attr1, Attr2
FROM Store
WINDOW 24 HOURS
```

f. Ordered Consume

```
SELECT FIRST(S.Attr1),
LAST(S.Attr1)
FROM Store S
WINDOW 24 HOURS
```

g. Eager Consume

```
SELECT Attr1, Attr2
FROM Store
WINDOW 24 HOURS
```

h. Sequential Read

```
SELECT Attr1, AVG(Attr2)
FROM Store
WINDOW 24 HOURS
GROUP BY Attr1
```

i. Random Read

```
SELECT S.Attr1
FROM Store S
WHERE PREV(S.Attr1) != S.Attr1
AND S.Attr1 < NEXT(S.Attr1)
```

j. Clustered Read

Figura 3.4: Esempi di query per ciascuna modalità di accesso.

Producer Update Pattern

Il produttore regola l'inserimento di nuovi valori all'interno di uno store. Al momento dell'inserimento dei nuovi valori i dati meno recenti possono scadere in diversi modi:

Never Expire: Il produttore inserisce continuamente dei dati che non scadono mai. Questo significa che i dati non vengono mai rimossi o migrati su disco. Un esempio di questa tipologia di query è riportato in figura 3.4.a: la query seleziona gli attributi 1 e 2 delle tuple in input e li inserisce all'interno dello store. Sarà quindi il consumatore a decidere quando i dati dovranno essere rimossi/migrati.

Ordered Expire: In questo caso i dati che diventano obsoleti sono quelli che vengono inseriti per primi nello store (FIFO). Il produttore specifica quindi una certa finestra temporale superata la quale i dati vengono rimossi/migrati (figura 3.4.b).

Unordered Expire: Questa modalità di aggiornamento prevede che i dati

che vengono 'marcati' come non più utili dipendono dal valore di alcuni attributi delle tuple in input. In pratica i dati non diventano obsoleti in maniera lineare ma possono venire marcati in maniera pseudo-casuale (RANDOM). Un esempio di questa modalità è descritta in figura 3.4.c.

Replaced Expire: La modalità Replaced Expire prevede che gli attributi delle tuple vengono continuamente aggiornati con nuovi valori. In pratica la quantità di tuple presenti nello store rimane costante ma i loro attribui vengono continuamente aggiornati. Ad esempio, se vogliamo tenere traccia del totale dei pedaggi pagati da un certo automobilista bisognerà aggiornare l'attributo 'bilancio' sommando di volta in volta i nuovi pagamenti. La figura 3.4.d mostra un esempio di questa query. La modalità Unordered Expire e quella Replaced Expire richiedono allo store di poter aggiornare le tuple sulla base dei valore di alcuni attributi (*value-based access*).

Consumer Update Patterns

Gli aggiornamenti degli store non vengono eseguiti soltanto in seguito all'inserimento di nuovi valori e quindi alla scadenza dei dati vecchi ma anche in seguito alla consumazione di alcune tuple da parte dei consumatori. I dati consumati non sono più di alcuna utilità e quindi vengono migrati o cancellati in modo da liberare memoria. La modalità con cui i dati vengono rimossi/migrati può essere *eagerly*, se le tuple vengono immediatamente migrate/rimosse non appena vengono marcate come 'consumate', o *lazily*, se le operazioni di migrazione/rimozione vengono eseguite periodicamente da un garbage collector.
I dati possono essere marcati come consumati in diversi modi:

Never Consume: In questo caso il consumatore non marca nessuna tupla come consumata e quindi sarà il produttore a gestire la rimozione delle tuple dalla memoria principale. Se anche il produttore è Never Expire, allora nessuna tupla verrà mai rimossa dalla memoria principale e quindi lo store sarà destinato a riempirsi completamente in breve tempo causando un *memory overflow* e degradando prestazioni e scalabilità del sistema. In tal caso converrà memorizzare tutte le tuple direttamente in memoria secondaria. Un esempio di query di questo tipo è riportato in figura 3.4.e.

Ordered Consume: In questa modalità il consumatore utilizza una finestra temporale in cui gli elementi che non rientrano in tale finestra vengono marcati

come consumati. In pratica i dati vengono consumati in maniera FIFO. (figura 3.4.f).

Eager Consume: In molti casi, il consumatore può seguire un approccio denominato *eager*, ovvero vengono marcati come consumati molti elementi all'interno della finestra. Ad esempio, quando una finestra diventa molto grande, il sistema può verificare che molti elementi all'interno di tale finestra non verranno utilizzati dagli operatori che agiscono su quella finestra e quindi vengono immediatamente marcati come consumati prima che la finestra proceda in avanti. Questo approccio non solo diminuisce il consumo di memoria, ma diminuisce il tempo necessario per elaborare una query in quanto valutare una query su una quantità di dati minore richiede meno tempo. Un esempio è riportato in figura 3.4.g, in cui si vede che la query seleziona solo il primo e l'ultimo elemento presenti nella finestra e quindi tutti gli altri possono essere immediatamente marcati come consumati dal sistema.

Store update pattern

La modalità con cui il consumatore e il produttore consumano ed eliminano gli elementi possono essere combinati in modo da assegnare allo store una sola modalità di aggiornamento dei dati (update pattern). Generalmente, la modalità di update dello store è dettata dal produttore. Questo perchè si vuole immediatamente eliminare dalla memoria primaria i dati scaduti (*expired*) e garantire al consumatore un accesso a delle informazioni accurate. Se il produttore ha però una modalità di aggiornamento dei dati di tipo Never Expire, allora l'update pattern dello store è dettato dall'update pattern del consumatore. In figura 3.5 vengono mostrate le varie modalità di aggiornamento adottate dallo store sulla base dell'update pattern del consumatore e del produttore.

	Never Consume	Ordered Consume	Eager Consume
Never Expire	NO-UPDATE	FIFO	FIFO/RANDOM
Ordered Expire	FIFO	FIFO	FIFO/RANDOM
Unordered Expire	RANDOM	RANDOM	RANDOM
Replaced Expire	IN-PLACE	IN-PLACE	IN-PLACE

Figura 3.5: Esempi di query per ciascuna modalità di accesso.

Ad eccezione del Replaced Expire, tutte gli aggiornamenti eseguiti dal produttore sono delle cancellazioni/migrazioni di dati obsoleti. Per quando riguarda l'aggiornamento di determinati valori viene utilizzata una modalità speciale di aggiornamento dei dati chiamata IN-PLACE. In totale sono state identificate 4 diverse modalità di aggiornamento dei dati:

- **NO-UPDATE**: Si ha questo update pattern quando il produttore è Never Expire e il consumatore è Never Consume, ovvero lo store non adotta nessun comportamento particolare per gestire i dati. In tal caso, conviene memorizzare i dati su disco in modo da evitare il memory overflow.

- **IN-PLACE**: Questa modalità si ha quando il produttore ha un update pattern di tipo Replaced Expire. Deve permettere di aggiornare i valori nello store sulla base di una certa chiave di ricerca. Se la chiave non esiste nello store, allora viene aggiunta.

- **RANDOM**: L'update pattern di tipo RANDOM può essere adottato in diverse situazione e permette di cancellare gli elementi nello store in maniera casuale.

- **FIFO**: La maggior parte delle applicazioni sfruttano questa modalità di aggiornamento dei dati. La modalità FIFO prevede che le cancellazioni avvengano in maniera ordinata procedendo dai dati piu vecchi fino a quelli piu recenti e favorisce le operazioni sulle code. Nel caso che il consumatore sia di tipo Eager Consume, allora si possono utilizzare sia la modalità RANDOM che quella FIFO in base alle caratteristiche del sistema. La modalità RANDOM diminuisce tipicamente il consumo di memoria perchè permette cancellazioni non ordinate di tipo 'eager', mentre la FIFO gestisce la memoria in maniera più efficiente come verrà descritto meglio nella sezione 3.2.7. Non c'è una scelta migliore, l'implementazione specifica deve essere calibrata sulla base dell'applicazione in esame.

3.2.5 Read pattern

La modalità con cui il consumatore richiede i dati da uno store definisce il read pattern. Dato che le finestre vengono solitamente memorizzate in memoria

principale, eseguire più volta una scansione lineare degli stessi dati non è una soluzione efficiente. In base al read pattern dei consumatori si possono eseguire delle ottimizzazioni, come ad esempio l'utilizzo di opportuni indici, in modo da migliorare e velocizzare le operazioni di ricerca.

Ci possono essere 3 modi con cui un consumatore può leggere i dati da uno store:

Sequenziale: Il consumatore accede ai dati nello store in maniera sequenziale. Viene specificata la finestra temporale e i dati all'interno di essa vengono acceduti uno dopo l'altro. Un esempio è riportato in figura 3.4.h.

Random: Il consumatore accede ai dati in maniera casuale, ovvero eseguendo dei 'salti' da un dato all'altro senza seguire un andamento sequenziale. Questo avviene generalmente quando vengono specificate delle condizioni nella clausola WHERE, oppure quando le tuple non rispettano lo stesso schema e si esegue una group-by. Un esempio è riportato in figura 3.4.i.

Clustered: In questa modalità di lettura, la richiesta è raggruppata intorno a certi elementi. Ad esempio, la query in figura 3.4.j, esegue delle letture con gli operatori 'next' e 'previous', il che comporta un accesso ai dati nell'intorno di alcuni valori.

3.2.6 Sharing e Access Pattern

Le analisi eseguite precedentemente sono state svolte considerando di avere un solo consumatore. Il sistema prevede però che più consumatori possano condividere le informazioni presenti in uno store e quindi ulteriori ottimizzazioni possono essere applicate. La figura 3.6 mostra un esempio di questo tipo.

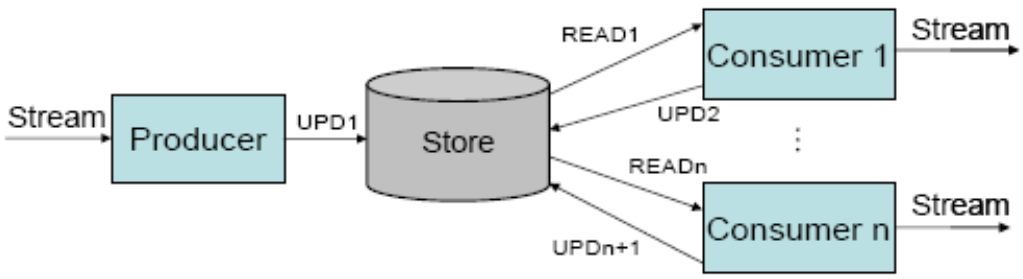


Figura 3.6: Esempio di store condiviso.

Per determinare l'update pattern di uno store condiviso bisogna analizzare come i consumatori accedono ai dati:

- Se almeno uno dei consumatori è di tipo Never Consume, allora l'insieme dei consumatori viene considerato Never Consume e lo store applica l'update pattern riportato in figura 3.5.

- Se tutti i consumatori sono Eager Consume, allora l'insieme dei consumatori può essere Eager Consume se si vuole risparmiare in memoria (RANDOM) oppure può anche essere Ordered Consume se non ci sono vincoli stringenti sulla memoria (FIFO).

- Tutti gli altri casi si considera l'insieme dei consumatori come un Ordered Consume update pattern.

La condivisione ha un impatto molto importante sulle performance. Gli operatori hanno bisogno solo di un certo sottoinsieme delle informazioni presenti nello store e conoscere in anticipo di quali dati necessitano gli operatori aiuta a velocizzare le operazioni di ricerca. Per ogni read pattern di ciascun consumatore devono essere previste delle strutture dati (indici) in grado di migliorare le operazioni di ricerca.

3.2.7 Strutture dati

Le strutture dati utilizzate dagli store vengono decise dinamicamente sulla base dei consumatori e produttori presenti su quello store. Queste strutture puntano a velocizzare l'accesso ai dati da parte dei consumatori e minimizzare lo spreco di memoria.
Ci sono fondamentalmente 4 operazioni di scrittura in qualunque update pattern: Inserimento, Cancellazione, Aggiornamento di un valore e Garbage collection. Il garbage collection è la procedura attraverso la quale si libera spazio dagli store in memoria eliminando/migrando i dati consumati od obsoleti su disco.
Di seguito analizziamo le strutture dati implementate dagli store sulla base dell'update pattern utilizzato (RANDOM, FIFO o IN-PLACE).

Implementazione di uno store RANDOM

La modalità RANDOM permette di cancellare qualsiasi elemento all'interno dello store in maniera non sequenziale.

La struttura dati più adatta per implementare uno store RANDOM è una **lista concatenata** in quanto risulta particolarmente flessibile e facilmente manipolabile in caso di cancellazioni ed inserimenti molto frequenti. Una lista concatenata aumenta o diminuisce le sue dimensioni in maniera dinamica ed in tempo costante.
Inserire ciascuna tupla entrante in un nuovo elemento della lista risulta però svantaggioso nel caso che i tassi di arrivo sia molto elevato, pertanto la lista non contiene direttamente le tuple ma contiene dei puntatori (*directors*) a dei blocchi che contengono i dati. Questo migliora le prestazioni in caso di inserimenti molto frequenti. Un blocco è implementato come un array di elementi con una particolare dimensione (fissa). Una nuova tupla entrante viene inserita nell'ultima posizione del blocco puntato dall'ultimo elemento della lista concatenata. Se tutti i blocchi sono pieni, viene creato un nuovo elemento in fondo alla lista e viene allocato un nuovo blocco di elementi in memoria.

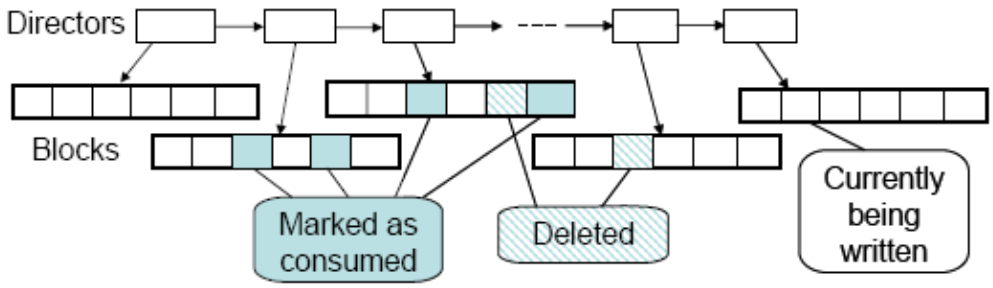


Figura 3.7: Struttura dati che implementa uno store RANDOM.

Quando un elemento viene consumato, la lista viene scandita fino ad arrivare al blocco che contiene quella tupla che verrà marcata come 'consumata' e quindi eliminata/migrata da parte del garbage collector. Tutte le richieste successive relative ad un dato 'consumato' falliranno. I dati scaduti (expired) perchè troppo vecchi e quelli consumati vengono marcati nello stesso modo e verranno allo

stesso modo migrati/eliminati dal garbage collector. Ci sono due modi con cui il garbage collector può cancellare i dati non più utili:

- Si attende che un blocco sia completamente pieno di elementi consumati (o expired). In tal caso il garbage collector del sistema eliminerà il puntatore a tale blocco dalla lista concatenata e il garbage collector del sistema operativo si occuperà di liberare la memoria.
- Gli elementi all'interno del blocco vengono eliminati non appena consumati. I blocchi decresceranno in dimensione fino a venire rimossi completamente. Questo approccio richiede maggiore lavoro da parte del garbage collector ma libera più velocemente la memoria.

Non è possibile riutilizzare i blocchi cancellati inserendoci nuovi valori provenienti dallo stream in quanto, essendo gli aggiornamenti/cancellazioni di tipo casuale, una scansione lineare della lista concatenata restituirebbe elementi che non sono ordinati temporalmente tra di loro (e valutare operatori di finestra temporale su dati non ordinati temporalmente richiederebbe di visitare tutti i dati all'interno dello store), pertanto i blocchi una volta che vengono cancellati non possono più essere riutilizzati e bisognerà allocare un nuovo blocco che verrà referenziato dall'ultimo elemento della lista concatenata. Nel caso di store condiviso, una tupla viene marcata come consumata quando è stata acceduta da tutti i consumatori. Comunque bisogna sempre garantire che un consumatore non possa accedere più volte allo stesso dato (ad esempio mantenendo traccia del mapping tra i consumatori e le tuple a cui hanno acceduto).

Implementazione di uno store FIFO

Uno degli aspetti negativi dell'implementare uno store in modalità RANDOM riguarda la gestione della memoria poco efficiente. Questo si verifica in seguito al fatto che i blocchi vengono continuamenti creati e rimossi, ma mai riutilizzati, facendo calare le performance. La necessità di poter accedere e cancellare i dati in maniera casuale impedisce di poter riutilizzare i blocchi consumati, in quanto, come spiegato precedentemente, una scansione lineare dei dati nella lista concatenata restituirebbe tuple non ordinate temporalmente tra di loro. La modalità FIFO supera questo problema permettendo di riutilizzare i blocchi completamente consumati inserendo nuovi dati. Questa caratteristica può essere

sfruttata grazie al fatto che la modalità FIFO è implementata da una **lista concatenata circolare** di puntatori a blocchi e che il produttore inserisce i dati in maniera sequenziale nella lista. Il risultato è che, anche se i dati vengono consumati in maniera casuale, una scansione lineare della lista fornisce sempre dati ordinati temporalmente tra di loro e quindi gli operatori di finestra temporale possono essere applicati senza la necessità di dover scorrere tutta la lista (evitando quindi di valutare tutti i dati presenti nello store). Ad esempio, se nella lista ci sono 5 directors che puntano a blocchi di tuple relative a 30 secondi di osservazioni, e la query mi richiede una finestra temporale di 10 secondi, io posso procedere nella lista dal puntatore del director attuale (che contiene le tuple più recenti) all'indietro fino a coprire una finestra di 10 secondi. Una volta coperta la finestra non ho piu necessità di guardare anche gli altri elementi perchè saranno sicuramente più vecchi di 10 secondi e quindi di nessuna utilità per quella query.

Nel caso di store condiviso, si possono eliminare tutte le tuple che non rientrano nella finestra temporale più grande. Basterà trovare il primo blocco che contiene delle tuple che non rientrano nella finestra temporale e cancellare tutti i blocchi ad esso precedenti nella lista.

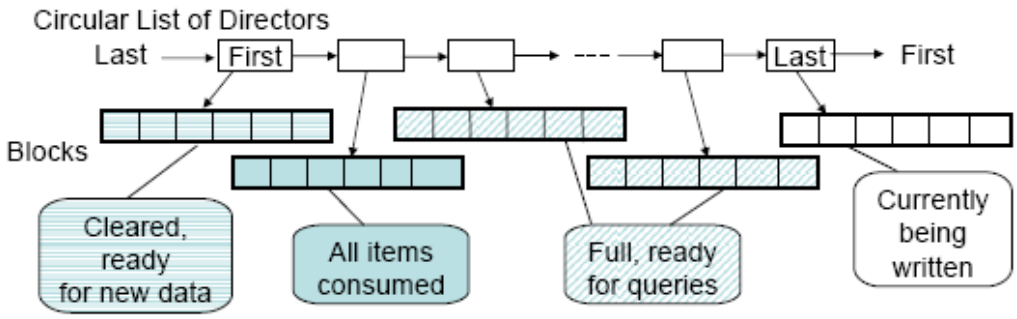


Figura 3.8: Struttura dati che implementa uno store FIFO.

Implementazione di uno store IN-PLACE

Nel caso di store IN-PLACE non ci sono mai cancellazioni, solo inserimenti o aggiornamenti di alcuni valori. Si può utilizzare un unico grande blocco per mantenere tutte le tuple e in tal caso le letture saranno molto veloci ma ci sarà la necessità di bloccare l'intero blocco ogni volta che bisogna eseguire un

aggiornamento. In un contesto dove il flusso di dati entranti è molto elavato rischierebbe di tenere bloccato il blocco molto spesso e quindi le prestazioni degraderebbero. In alternativa, si possono utilizzare diversi blocchi in modo da ridurre la congestione (simile al partizionamento orizzontale dei database che divide una tabella in più tabelle).
L'implementazione di uno store IN-PLACE prevede l'esistenza di un indice per recuperare rapidamente gli elementi che devono essere aggiornati.

3.2.8 Indici

L'utilizzo di un indice migliora generalmente il tempo di ricerca dei dati ma aumenta l'utilizzo di memoria e rende più lente le operazioni di modifica o inserimento, pertanto l'utilizzo di un indice deve essere valutata in base all'applicazione in esame.
Nel caso di store IN-PLACE il problema più grande risiede nel fatto che per ogni richiesta bisogna scorrere tutti gli elementi del blocco in maniera sequenziale, il che rende la ricerca dei valori poco performante. La soluzione migliore risiede nella costruzione di un indice che tenga conto dei parametri richiesti dalla query. In questo caso lo schema diventa un parametro fondamentale. Nel caso si voglia aggiornare un valore, l'accesso *value-based* prevede l'esistenza di una struttura particolare chiamata *value-based index* (indice basato sul valore). Per la costruzione di questo indice lo store manager valuta la clausola WHERE della query in esame. Come primo passo, i parametri specificati nella clausola vengono tradotti in uno schema ben preciso (*index schema*) e vengono valutati gli operatori logici applicati a tali parametri (minore, maggiore, ecc.). Ad esempio, considerando la query: dammi i pedaggi pagati al minuto '*min*' sulla strada '*street*' in direzione '*dir*'. In questo caso, lo schema sarà composto da 3 coppie con la seguente struttura: (`(min, eq), (street, eq), (dir, eq)`). L'indice sarà costruito tenendo conto di questi attributi. Quando arriva una richiesta con lo stesso schema viene consultato l'indice e i valori vengono ritornati.

Per quanto riguarda gli store RANDOM e FIFO le ricerche vengono ottimizzate sfruttando sia la cache sia particolari indici chiamati *id-index*. Quando una tupla viene richiesta, si controlla se il blocco che contiene la tupla è in cache. Se è presente allora viene restituito senza bisogno di recuperarlo dallo store, altrimenti (*cache miss*) bisognerà recuperare il blocco dalla lista concatenata. Il

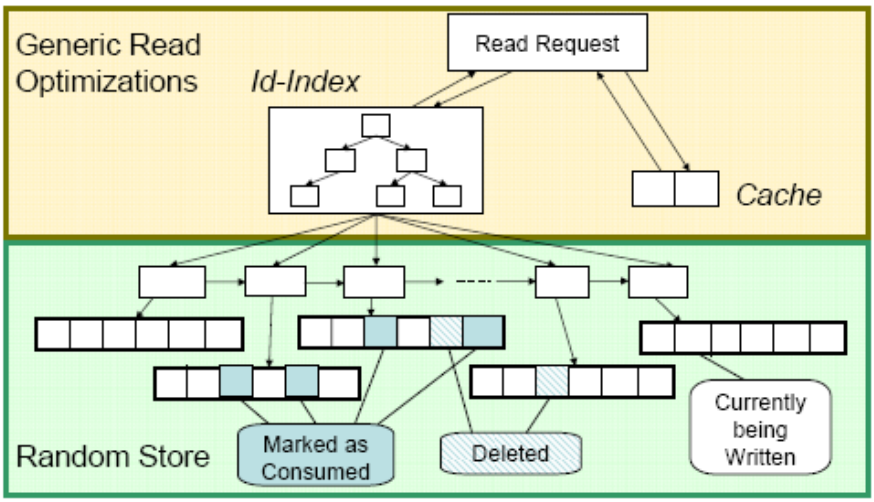


Figura 3.9: Costruzione dell'indice sui dati.

vantaggio nello sfruttare la cache risiede nel fatto che letture sequenziali cadranno con grande probabilità nello stesso blocco e quindi potrano essere restituite senza bisogno di interrogare lo store, migliorando notevolmente i tempi di risposta.
L'id-index è un indice ad albero costruito sulla lista concatenata e migliora la velocità delle ricerche in quanto evita di dover scorrere tutta la lista per cercare i valori. Se le letture sono sempre sequenziali (sequential read), allora l'utilizzo dell'indice è marginale e non migliora le performance, in quanto i blocchi vengono acceduti sempre sequenzialmente e non c'è necessità di consultare l'indice. La velocità delle ricerche casuali (random read) è invece notevolmente migliorata dalla presenza dell'indice, in quanto la cache viene raramente sfruttata e per recuperare il blocco opportuno si evita di dover scorrere tutta la lista. L'indice viene sempre costruito sulla base dello schema delle richieste in arrivo e quindi è di fondamentale importanza che i dati rispettino lo stesso schema.
Infine, per quanto riguarda le letture clustered (clustered read) l'accesso è pseudorandom e sfrutta un principio di località. Conviene quindi costruire un indice che tenga conto anche della cache in quanto richieste sequenziali hanno una grande probabilità di cadere nello stesso blocco e quindi di essere presenti in cache.

3.3 Query Processing Engine (QPE)

Il Query Processing Engine (QPE) è quella parte del DSMS che si occupa di ricevere le query da eseguire, ottimizzarle e inviare le richieste allo store manager. In questa sezione si focalizzerà l'attenzione sugli aspetti di cui un Query Processing Engine deve tenere conto. In particolare verrà descritta la semantica delle query continue, i linguaggi, gli operatori e infine delle ottimizzazione che potranno essere applicate per migliorare i tempi di risposta, come l'adattamento dinamico del query plan al data stream e gli algoritmi di query approximation.

3.3.1 Semantica e sintassi delle query continue

Le query continue sono una classe di query molto importante e molto richiesta da un gran numero di applicazioni. Molti linguaggi sono stati proposti di recente ma il problema è che non è ancora nota una semantica ben precisa per poter definire una query continua. La maggior parte dei linguaggi propongono una semantica informale, spesso basata solamente su esempi.

Semantica

La semantica delle query continue può risultare molto complessa, ed esprimere una qualsiasi tipologia di query se la semantica non è ben definita può risultare poco chiaro. Una definizione formale assegna ad una query espressa in qualsiasi linguaggio un significato ben preciso. Risulta molto utile in quanto permette di descrivere le azioni di una query in maniera non ambigua, indipendente dalla macchina e dal linguaggio, e permette di determinare, data una query sintatticamente corretta ed un valido input per essa, l'output che verrà prodotto. Lo studio della semantica per le query continue permette quindi di evidenziare ambiguità e complessità non esplicite e fornire una base per l'implementazione, l'analisi e la verifica delle query (in particolare con dimostrazioni di correttezza). Per questo motivo è bene studiare una semantica formale per le query continue in un dato **linguaggio generico** [6].

Il linguaggio generico in esame è denotato con la lettera L e contiene due tipologie di dati: *streams* e *relations*. Gli stream e le relation hanno associato uno *schema*, il quale contiene una serie di attributi. Stream e relations sono definiti in un dominio temporale discreto T. Un *istante temporale* è un certo

valore appartenente a T. In pratica, T può essere visto come l'insieme di numeri interi $\{0, 1, 2, ...\}$, dove 0 rappresenta l'istante iniziale.

Stream: Uno stream S è un insieme, anche infinito, di elementi $\langle s, t \rangle$, dove s è una tupla con lo stesso schema di S e $t \in T$ è il timestamp.

Relation: Una relazione R è un mapping da T ad un insieme di tuple con lo stesso schema di R.

In pratica uno stream è una collezione di tuple con un certo timestamp. Intuitivamente, l'elemento $\langle s, t \rangle$ di uno stream S indica ce la tupla s è arrivata da S al tempo t. Una relazione invece la si può vedere come una tabella di un database tradizionale in cui però è espresso anche il parametro temporale, infatti $R(t)$ denota l'insieme di tuple nell'istante temporale t.

Le query espressa nel linguaggio generico L sono composte da operatori appartenenti a tre classi: *stream-to-relation*, *relation-to-relation* e *relation-to-stream*.

- L'operatore stream-to-relation prende in input uno stream e produce una relazione. L'insieme delle tuple nella relazione in output all'istante t dipende dall'insieme di tuple in input che si presentano negli istanti $\leq t$.

- L'operatore relation-to-relation prende in input una o più relazioni e produce un altra relazione come output. L'insieme delle tuple nella relazione in output all'istante t dipende dall'insieme di tuple nella relazione in input all'istante t.

- L'operatore relation-to-stream prende in input una relazione e produce uno stream. L'elemento nello stream in output con timestamp t dipende solamente dallo stato della relazione in input negli istanti $\leq t$.

Concettualmente, le query o gli operatori girano continuamente all'interno del dominio temporale. Quando il tempo avanza da $t - 1$ a t, la query o gli operatori producono un nuovo output corrispondente a t.

Il linguaggio generico non specifica come questi operatori funzionano, sono delle *black box* che rispettano le proprietà viste sopra. Per avere una implementazione concreta di questo linguaggio si può analizzare il linguaggio CQL

(*Continuos Query Language*). In CQL, l'operatore stream-to-relation è implementato tramite l'operatore *sliding window* (finestra scorrevole). Gli operatori relation-to-relation in CQL sono gli operatori standard del SQL (operatori che agiscono sulle tabelle). Per quanto riguarda gli operatori relation-to-stream, ce ne sono di 3 tipi: *Istream*, che inserisce in uno stream tutte le tuple presenti nella relazione ad un certo istante temporale, *Dstream*, che inserisce in uno stream tutte le tuple che sono state cancellate da una relazione ad un certo istante temporale, e *Rstream*, che inserisce nello stream tutte le tuple presenti nella relazione in qualsiasi istante temporale.

Sintassi

La figura 3.10 presenta una sintassi astratta per il linguaggio generico L usando il formalismo Backus-Naur Form (BNF). La BNF è una metasintassi, ovvero un formalismo attraverso il quale è possibile descrivere la sintassi di linguaggi formali. Questa sintassi è specificata al solo scopo di spiegare meglio la semantica del linguaggio.
Per la specifica della semantica vengono utilizzati i seguenti domini:

- *Time domain* (T): $T = 0, 1, ...$ rappresenta il dominio temporale discreto.
- *Tuple domain* (TP): Dominio delle tuple. Una tupla è una sequenza finita di valori atomici. Non si fanno distinzioni tra le tuple con schema differente.
- *Tuple multiset domain* (Σ): Rappresenta l'insieme finito di gruppi di tuple.
- *Relation domain* (R): $R = T \rightarrow \Sigma$ è il dominio di funzioni che mappano un istante temporale ad un insieme di tuple.
- *Stream domain* (S): Il dominio, anche infinito, di tutti gli stream su $TP \times T$.
- *Relational operator domain* (R_{op}): $R_{op} = \Sigma \times ... \times \Sigma \rightarrow \Sigma$, è il dominio di funzioni che producono un insieme di tuple da uno o più insiemi di tuple. Le normali query SQL appartengono a questo dominio.
- *Syntactic domains*: Il dominio dei termini sintattici elencati in figura 3.10. Ad esempio, *Query* è il dominio delle query continue sintatticamente corrette, mentre *R2ROp* rappresenta il dominio degli operatori relation-to-relation.

$$
\begin{array}{rcl}
\mathtt{Q} & ::= & \mathtt{Q_R} \mid \mathtt{Q_S} \\
\mathtt{Q_R} & ::= & \mathtt{RName} \mid \mathtt{R2R\text{-}Op}(\mathtt{Q_R^1}, \ldots, \mathtt{Q_R^n}) \mid \mathtt{S2R\text{-}Op}(\mathtt{Q_S}) \\
\mathtt{Q_S} & ::= & \mathtt{SName} \mid \mathtt{R2S\text{-}Op}(\mathtt{Q_R}) \\
\mathtt{RName} & ::= & \mathtt{Id} \\
\mathtt{SName} & ::= & \mathtt{Id}
\end{array}
$$

Symbol	Description	Domain
Q	Continuous Query (CQ) in $\mathcal{L}$	*Query*
$\mathtt{Q_R}$	CQ producing a relation	*RelQuery*
$\mathtt{Q_S}$	CQ producing a stream	*StrQuery*
S2R-Op	Stream-to-Relation Operator	*S2ROp*
R2R-Op	Relation-to-Relation Operator	*R2ROp*
R2S-Op	Relation-to-Stream Operator	*R2SOp*
RName	Relation Name	*Identifier*
SName	Stream Name	*Identifier*
Id	Identifier	*Identifier*

Figura 3.10: Sintassi astratta per L. Descrizione dei simboli e dei domini.

- *Relation Lookup domain (RelLookup)*: Dominio di funzioni che mappano gli identificativi (nome della relazione) con le rispettive relazioni. *Identifier* $\rightarrow R$.

- *Stream Lookup domain (StrLookup)*: Dominio di funzioni che mappano gli identificativi (nome di uno stream) con il rispettivo stream. *Identifier* $\rightarrow S$.

Semantica denotazionale

La semantica denotazionale è un approccio teso a formalizzare il significato dei linguaggi di programmazione (o di query), costruendo oggetti matematici (chiamati denotazioni) i quali descrivono i significati delle espressioni del linguaggio. Contrariamente alla semantica operazionale, l'interesse viene posto esclusivmente sull'*effetto dell'esecuzione* della query. Per effetto si intende l'as-

sociazione tra lo stato iniziale e quello finale. L'idea è di definire una funzione semantica che leghi ogni costrutto sintattico ad una funzione matematica. La semantica denotazionale fornisce la denotazione (significato) ad una query come una funzione che associa l'input all'output. Nel nostro caso l'input e l'output possono essere uno stream, una relazione ed un istante temporale. La funzione semantica viene anche chiamata *funzione significato* o *meaning function* e si indica con la lettera M. M quindi mappa una query espressa in un linguaggio nella sua funzione di input-output. La tabella 3.1 elenca le meaning function utilizzate. M è la meaning function principale ed assegna un significato preciso alla query nel suo complesso. Le altre meaning function assegnano un significato alle sottocomponenti delle query; ad esempio, M_{R2R} mappa un operatore di tipo relation-to-relation ad una funzione che opera sulle relazioni tradizionali. Il significato della query si determina per composizione della semantica delle sue componenti. In pratica, ad ogni costrutto sintattico (parte della query) C viene associata la sua denotazione (significato) $[\![C]\!]$, un oggetto matematico che rappresenta il contributo di C al significato di una qualunque query che contenga C. Le parentesi $[\![]\!]$ sono chiamate parentesi semantiche ed esprimono la denotazione di una certa sintassi e sono di fatto una funzione matematica.

Query component	**Meaning Function**	**Signature**
Q	M	$Query \rightarrow (RelLookup \times StrLookup \times T \rightarrow (\Sigma \cup S))$
Q_R	M_R	$RelQuery \rightarrow (RelLookup \times StrLookup \times T \rightarrow \Sigma)$
Q_S	M_S	$RelQuery \rightarrow (RelLookup \times StrLookup \times T \rightarrow S)$
$S2R - Op$	M_{S2R}	$S2ROp \rightarrow (S \times T \rightarrow \Sigma)$
$R2R - Op$	M_{R2R}	$R2ROp \rightarrow R_{op}$
$R2S - Op$	M_{R2S}	$R2SOp \rightarrow (S \times T \rightarrow S)$

Tabella 3.1: Meaning function nella semantica denotazionale.

Ciascuna meaning function viene specificata tramite le regole BNF. Ad esempio, M viene specificata in due parti: la prima corrispondente alla derivazione di Q da Q_R e la seconda corrispondente alla derivazione di Q da Q_S.

Analizziamo le meaning function:

- M : La funzione di input-output $M[\![Q]\!]$ prodotta da M per la query Q prende in input tre parametri: i primi due sono funzioni che mappano (referenziano) il nome dello stream o della relazioni specificati nella query allo stream o relazione effettiva, l'ultimo parametro è l'istante temporale. $M[\![Q]\!](r, s, t)$ specifica l'output prodotto da Q all'istante temporale t. $M[\![Q]\!](r, s, t)$ invoca $M_R[\![Q_R]\!](r, s, t)$ se $Q = Q_R$ e produce una relazione come output, oppure invoca $M_S[\![Q_S]\!](r, s, t)$ se $Q = Q_S$ e produce uno stream come output. Quindi abbiamo che:

$$M[\![Q_R]\!] = \lambda_r.\lambda_s.\lambda_t.M_R[\![Q_R]\!](r, s, t)$$
$$M[\![Q_S]\!] = \lambda_r.\lambda_s.\lambda_t.\{\langle e, t'\rangle : \langle e, t'\rangle \in M_s[\![Q_S]\!](r, s, t) \wedge t' = t\}$$

  dove λ rappresenta il lambda-calcolo, ovvero $\lambda x.t$ rappresenta la funzione che, quando applicata ad un argomento v, produce t in cui v rimpiazza le occorrenze di x in t. Ad esempio, $\lambda x.x + 1$ rappresenta la funzione successore mentre $\lambda x.x$ rappresenta la funzione identità.

- M_R: Se Q_R è una sottoparte di una query che produce una relazione, $M_R[\![Q_R]\!](r, s, t)$ specifica l'insieme di tuple nella relazione in output al tempo t. I parametri r ed s sono le funzioni che mappano lo stream o la relazione nello stream o relazione effettivamente presente. M_R richama ricorsivamente sè stessa, M_S, M_{R2R}, M_{S2R} e/o r, a seconda della struttura di Q_R. Ad esempio, se $Q_R = RName$, allora $M_R[\![Q_R]\!](r, s, t)$ usa la funzione r per cercare la relazione variabile nel tempo corrispondente al nome $RName$, e cerca l'insieme di tuple che identificano la relazione all'istante temporale t.

$$\begin{aligned}
M_R[\![Rname]\!] &= \lambda_r.\lambda_s.\lambda_t.r(RName)(t) \\
M_R[\![R2R\text{-}Op(Q_R^1, ..., Q_R^n)]\!] &= \lambda_r.\lambda_s.\lambda_t.M_{R2R}[\![R2R\text{-} \\
&\quad Op]\!](M_R[\![Q_R^1]\!](r, s, t), ..., M_R[\![Q_R^n]\!](r, s, t)) \\
M_R[\![S2R\text{-}Op(Q_S)]\!] &= \lambda_r.\lambda_s.\lambda_t.M_{S2R}[\![S2R\text{-} \\
&\quad Op]\!](M_S[\![Q_S]\!](r, s, t), t)
\end{aligned}$$

- M_S: Se Q_S è una sottoparte di una query che produce uno stream, $M_S[\![Q_S]\!](r, s, t)$ è l'insieme di tuple nello stream in output con il timestamp $\leq t$. La definizione di $M_S[\![Q_S]\!]$ nel caso di $Q_S = R2S\text{-}Op(Q_R)$, il lambda calcolo di $\lambda t'.M_R[\![Q_R]\!](r, s, t')$ definisce una funzione che prende un singolo

parametro t' e ritorna l'insieme di tuple al tempo t' nella relazione prodotta dalla sottoquery Q_R.

$$\begin{aligned} M_R[\![Sname]\!] &= \lambda_r.\lambda_s.\lambda_t.\{\langle e,t'\rangle : \langle e,t'\rangle \in s(SName) \wedge t' \leq t\} \\ M_S[\![R2S\text{-}Op(Q_R)]\!] &= \lambda_r.\lambda_s.\lambda_t.M_{R2S}[\![R2S\text{-}Op]\!]((\lambda t'.M_R[\![Q_R]\!](r,s,t')),t) \end{aligned}$$

Rappresentiamo ora la sintassi astratta utilizzando le regole BNF per gli operatori analizzati sopra:

$$\begin{aligned} S2R\text{-}Op &::= \texttt{Now} \mid \texttt{Range(T)} \mid \texttt{Row(N)} \\ R2R\text{-}Op &::= \texttt{SemiJoin(i,j)} \mid \texttt{Filter(i,v)} \\ R2S\text{-}Op &::= \texttt{IStream} \mid \texttt{DStream} \mid \texttt{RStream} \end{aligned}$$

Tabella 3.2: Sintassi astratta per gli operatori del linguaggio CQL.

Per concludere, rappresentiamo la semantica formale per gli operatori in tabella 3.2 e definiamo le meaning function M_{S2R}, M_{R2R} e M_{R2S}.

- M_{S2R}: Nel caso di operatori stream-to-relation ($S2R\text{-}Op$) consideriamo tre tipi diversi di finestre scorrevoli: Now restituisce le tuple di S con il timestamp t (il timestamp più recente), $Range$ prende in input un parametro T e restituisce le tuple nel range $[t-T,t]$, Row prende un intero N e restituire le N tuple più recenti con il timestamp $\leq t$.

$$\begin{aligned} M_{S2R}[\![\texttt{Now}]\!] &= \lambda_S.\lambda_t.\{e : \langle e,t\rangle \in S\} \\ M_{S2R}[\![\texttt{Range}(T)]\!] &= \lambda_S.\lambda_t.\{e : \langle e,t'\rangle \in S \wedge max(t-T,0) \leq t' \leq t\} \\ M_{S2R}[\![\texttt{Range}(N)]\!] &= \lambda_S.\lambda_t.\{e : \langle e,t'\rangle \in S \wedge (t' \leq t) \wedge (N \geq \\ &\quad |\{\langle e,t''\rangle \in S : t' \leq t'' \leq t\}|)\} \end{aligned}$$

- M_{R2R}: Nel caso di operatori relation-to-relation ($R2R\text{-}Op$) consideriamo due tipi di operazioni possibili: $\texttt{SemiJoin}(i,j)$ che esegue un semijoin tra l'i-esimo attributo della prima relazione con il j-esimo elemento della seconda relazione (restituisce l'insieme di tuple in cui i valori degli attributi scelti sono uguali), $\texttt{Filter}(i,v)$ che restituisce tutte le tuple che hanno il valore v nell'i-esimo attributo.

$$\begin{aligned} M_{R2R}[\![\texttt{SemiJoin}(i,j)]\!] &= \lambda_{E_1}.\lambda_{E_2}.\{e_1 : e_1 \in E \wedge (\exists e_2 \in E_2 : e_1.i = \\ &\quad e_2.j)\} \\ M_{R2R}[\![\texttt{Filter}(i,v)]\!] &= \lambda_E.\{e : e \in E \wedge e.i = v\} \end{aligned}$$

- M_{R2S}: Nel caso di operatori relation-to-stream ($R2S$-Op) ci sono tre tipi di operazioni: `IStream` che prende una relazione R e un istante temporale t e produce uno stream di tutte le tuple che sono state inserite in R al tempo t (si indice con $R(t)$), `DStream` che inserisce in uno stream le tuple che sono state cancellate da R al tempo t, ovvero le tuple che erano presenti in R al tempo $t-1$, `RStream` che inserisce nello stream tutte le tuple presenti nella relazione a qualsiasi istante temporale.

$$
\begin{aligned}
M_{R2S}[\![\texttt{IStream}]\!] &= \lambda_R.\lambda_t.\{e : \langle e, t' \rangle : t' \leq t \wedge e \in R(t) \wedge e \notin R(t-1)\} \\
M_{R2S}[\![\texttt{DStream}]\!] &= \lambda_R.\lambda_t.\{e : \langle e, t' \rangle : t' \leq t \wedge e \in R(t-1) \wedge e \notin R(t)\} \\
M_{R2S}[\![\texttt{RStream}]\!] &= \lambda_R.\lambda_t.\{e : \langle e, t' \rangle : t' \leq t \wedge e \in R(t)\}
\end{aligned}
$$

3.3.2 Linguaggi per le query continue

Analizziamo ora le varie tipologie di linguaggi per poter esprimere query continue. Fino ad oggi sono stati proposti tre paradigmi diversi per l'interrogazione di dati in streaming: Relation-based, Object-based e Procedurali.

Linguaggi Relation-based

Sono presenti tre linguaggi relation-based differenti per esprimere delle query continue e sono il CQL, StreaQuel e AQuery. Ognuno di essi è basato sull'SQL (SQL-like), quindi sono linguaggi dichiarativi basati sul calcolo relazionale, e supportano gli operatori di finestra e ordinamento. I linguaggi dichiarativi, a differenza di quelli procedurali che si analizzeranno in seguito, specificano direttamente le caratteristiche che l'output deve avere ma non come ottenerlo. CQL (Continuous Query Language) è utilizzato nel sistema STREAM e considera gli stream e le finestre come relazioni ordinate dal timestamp. Gli operatori relation-to-stream permettono di convertire i risultati delle query in stream. Con questo operatore gli utenti possono esplicitamente specificare la semantica della query, come definito nelle equazioni 3.1 e 3.2, dove $A(Q,t)$ rappresenta l'insieme delle risposte ad una query Q al tempo t, τ è il tempo corrente e 0 è il tempo di partenza:

$$A(Q,\tau) = \bigcup_{t=1}^{\tau} (A(Q,t) - A(Q,t-1)) \cup A(Q,0) \tag{3.1}$$

$$A(Q,\tau) = \bigcup_{t=0}^{\tau} A(Q,t) \tag{3.2}$$

Se la query continua è monotona (accesso sequenziale ai valori) sarà sufficiente rivalutare la query sui nuovi dati arrivati e aggiungere i nuovi risultati all'insieme di tuple in output (3.1), altrimenti, se la query non è monotona (accesso random) bisognerà ricalcolare dall'inizio tutti i valori (3.2).
Inoltre, nei linguaggi relazionali, bisognerà specificare esplicitamente il *sampling rate* (velocità di campionamento dello stream continuo in input), ad esempio con il comando `10 % SAMPLE`.

StreaQuel, il linguaggio utilizzato in TelegraphCQ, permette di utilizzare delle finestre molto sofisticate e non richiede nessun operatore di tipo relation-to-stream per considerare gli output e gli input come stream. Ciascuna query espressa in StreaQuel è seguita da un costrutto `for-loop` che itera su una variabile t. Il ciclo contiene un comando `WindowIs` che specifica il tipo e la dimensione della finestra. Ad esempio, consideriamo uno stream `S`. `NOW` rappresenta il tempo corrente e vogliamo specificare una certa finestra scorrevole su `S` di dimensione `5` unità temporali. In tal caso dopo la query bisognerà aggiungere il seguente ciclo:

```
for(t=NOW; t<NOW+50; t++)
    WindowIs(S, t-4, t)
```

AQuery utilizza l'algebra sulle query e un linguaggio SQL-based per gestire i dati ordinati. Le colonne delle tabelle sono gestite come array in cui l'ordine è fondamentale e in cui sono ammesse operazioni del tipo: `next`, `previous`, `first` e `last`. Ad esempio, una query continua in grado di evidenziare le differenze di prezzo da uno stream continuo di quotazioni di borsa può essere espresso nel seguente modo:

```
SELECT price - prev(price)
FROM Trades WHERE company = 'IBM'
```

Linguaggi Object-based

Uno degli approcci maggiormente usati dai linguaggi object-based consiste nel classificare gli elementi provenienti dallo stream. Questo metodo è utilizzato dalla rete di monitoraggio Tribeca il cui obiettivo è leggere i dati provenienti da una certa sorgente (ad esempio i dati in arrivo real time da una rete) ed eseguire un certo insieme di query compilate. In particolare, Tribeca usa i DDL (Data Definition Language) per la definizione (o modifica) degli oggetti (ad esempio, per categorizzare i differenti pacchetti che circolano sulla rete).

Un altro modo per gestire come oggetti i dati provenienti da uno stream è utilizzare gli ADT (Abstract Definition Language). Questo metodo è utilizzare dal sistema COUGAR che si occupa di gestire i dati provenienti da una sensor network. I sensori vengono modellati tramite ADT in cui l'interfaccia è data dai metodi che processano i dati in arrivo da quel sensore. Il linguaggio di query proposto è basato su una sintassi SQL-like e include un comando del tipo `every()` che indica la frequenza con cui la query viene rivalutata.

Linguaggi Procedurali

Un alternativa ai linguaggi di query dichiarativi, in cui l'utente specifica COSA vuole e si disinteressa di COME la computazione avvenga (linguaggio SQL), sono i linguaggi procedurali. Nei linguaggi procedurali l'utente specifica direttamente come i dati vengono manipolati per ottenere il risultato desiderato tramite una serie di istruzioni. E' bene sottolineare che qualsiasi interrogazione può essere espressa sia come algebra relazionale che come calcolo relazionale, ottenendo lo stesso risultato. L'algebra relazionale appartiene alla famiglia dei linguaggi procedurali, in cui gli operatori agiscono su relazioni (insiemi di tuple con certi attributi) e producono relazioni. Gli operatori dell'algebra relazionale sono: unione, intersezione, differenza, selezione, proezione e join. Ad esempio, nel linguaggio procedurale di Aurora, l'utente costruisce direttamente un query plans tramite interfaccia grafica (nei sistemi dichiarativi è il sistema che in automatico crea il query plan) e specifica gli operatori, rappresentati tramite 'scatole' (box), le join da eseguire, unendo l'uscita di diverse scatole in un unica freccia, ecc. Il sistema si occuperà solamente di ottimizzare tale query plan, ad esempio cambiando la disposizione delle scatole o aggiungendo e rimuovendo operatori. Aurora inoltre fornisce diversi operatori che non sono previsti in altri linguaggi:

map che applica una certa funzione a ciascun elemento (simile all'operatore `each` del linguaggio AQuery), *resample* che interpola i valori mancanti in una certa finestra e *drop* che a random scarta degli elementi quando l'input rate è troppo elevato.

3.3.3 Operatori per il data streaming

I problemi di ricerca che riguardano l'estrazione di conoscenza e l'interrogazione di stream di dati continui hanno creato nuove soluzioni che possono essere classificate in due categorie: *Data-based* e *Task-based*. La difficoltà maggiore risiede nel fatto che molti problemi sono computazionalmente complessi e i sistemi che gestiscono in real-time flussi di dati continui devono gestire una mole di dati potenzialmente infinita con vincoli stringenti sulla capacità di memorizzazione primaria. Le soluzioni data-based a questi problemi puntano ad esaminare solo un sottoinsieme dei dati totali, ad esempio partizionando orizzontalmente o verticalmente, in modo da rendere il problema computabile. D'altra parte, le soluzioni task-based provengono dalla teoria della complessità computazionale e puntano a trovare algoritmi che riescano a calcolare la soluzione in maniera efficiente dal punto di vista del tempo e dello spazio.

Tra le tecniche data-based si possono trovare: sampling, load shedding, sketching, synopsis e aggregation. Invece, tra le tecniche task-based si possono trovare: algoritmi di approssimazione, sliding window e data stream mining [20].

Tecniche diverse possono essere combinate tra loro, ad esempio gli algoritmi di approssimazione possono sfruttate il sampling o i synopsis per diminuire la quantità di dati su cui applicare l'algoritmo. In questa sezione si discuteranno meglio queste tecniche. In particolare, si dettaglieranno le tecniche che permettono di eseguire gli operatori bloccanti, come l'approssimazione, il data stream mining e le finestre scorrevoli.

Operatori non bloccanti

Gli operatori bloccanti sono solitamente degli operatori che richiedono di processare l'intero input prima che un qualsiasi output venga prodotto. Ad esempio, gli operatori *Count*, *Min*, *Max*, *Sort* e *Avg* richiedono di valutare tutti i dati prima di poter produrre una risposta corretta. Il problema nei DSMS è dato

dal fatto che il flusso di dati entrante potrebbe essere potenzialmente infinito e quindi non può essere memorizzato interamente per potergli applicare gli operatori. Esistono fondamentalmente tre diverse tecniche per permettere la valutazione di operatori bloccanti su stream infiniti: *Windowing*, *Approximation* ed *Incremental Evaluation*.

Qualsiasi operatore può essere reso non bloccante restrigendo il range di valutazione su una finestra di dimensione limitata in grado di poter essere mantenuta in memoria principale (windowing). Inoltre, per evitare la rivalutazione dell'intera finestra ogni volta che si presenta un nuovo elemento, gli operatori possono essere resi 'incrementali'. Ad esempio, operatori come l'`AVERAGE` possono essere incrementalmente aggiornati mantenendo una somma cumulativa di tutti i valori e del numero di elementi presenti (incremental evaluation).

Un altra tecnica per 'sbloccare' un operatore bloccante è sfruttare la conoscenza che si ha sui dati. Utilizzare, ad esempio, dei vincoli sullo schema dei dati può essere sfruttato per inserire all'interno dello stream una o più tuple di *marking* con lo stesso schema dello stream di dati che serve per comunicare al DSMS che le tuple ricevute fino a quel punto possono essere valutate dall'operatore. Tale tecnica prende il nome di *punctuations*. La punctuations non da nessuna garanzia sulle caratteristiche dello stream di dati che segue la tupla di marking, in pratica è un indicatore di fine substream (EOSS, end of a substream). Gli operatori quindi attendono fino all'arrivo della tupla di EOSS e poi vengono valutati. Questa tecnica è molto simile alle tecniche di windowing con la differenza che la dimensione dell'insieme di dati su cui valutare gli operatori non viene decisa dalla destinazione ma dalla sorgente che inserisce opportunamente delle tuple di marking nel flusso di dati.

Algoritmi di approssimazione

Una delle tecniche per rendere non bloccanti degli operatori bloccanti riguarda l'utilizzo di algoritmi di approssimazione [24]. Questi algoritmi sono stati pensati per risolvere in maniera efficiente problemi computazionalmente difficili (in questo caso la valutazione di una query su una quantità di dati potenzialmente infinita) e forniscono soluzioni approssimate con un certo margine di errore. Questi algoritmi puntano inizialmente a costruire dei *synopses*, che contengono dei 'riassunti' su un certo numero di dati dello stream, e successivamente le query vengono valutate su questi riassunti. Bisognerà gestire il trade-off tra accuracy e

utilizzo di memoria, in quanto meno compatto è il riassunto tanto maggiore sarà l'accuracy e viceversa tanto maggiore sarà la compressione delle informazioni tanto minore sarà la precisione delle risposte. Gli algoritmi di approssimazione su stream infiniti possono essere suddivisi in diverse categoria sulla base delle modalità di creazione dei synopses:

- *Counting methods*: Questi algoritmi sfruttano informazioni statistiche per descrivere il flusso di dati, come i quantili e le frequenze di certi elementi. Queste misure vengono sempre fatte tenendo conto di un certo margine di errore.

- *Hashing methods*: Vengono principalmente usati per contare o campionare gli elementi provenienti dallo stream. Ad esempio, possono essere sfruttati per ricercare gli elementi più frequenti nello stream.

- *Sampling methods*: Memorizzano gli elementi provenienti da uno stream con una certa probabilità. Non sono sempre applicabili per tutte le query di aggregazione, ad esempio quando si cerca il valore massimo in uno stream questi metodi non possono essere applicati perchè bisogna poter valutare tutti gli elementi. Questi metodi possono però essere applicati per molte

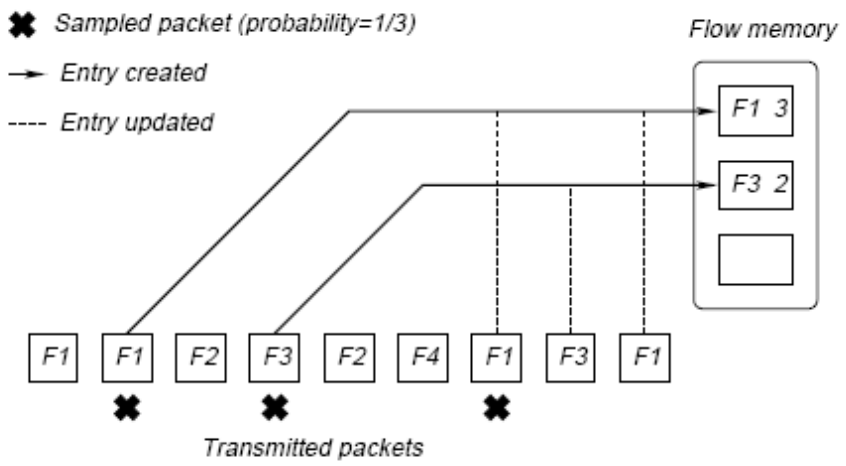


Figura 3.11: Sampling method.

query di aggregazione (`COUNT`, `AVG`, ecc) tenendo conto di un certo margine di errore rispetto al valore reale. Un esempio è riportato in figura 3.11.

- *Sketches*: Vengono usati in molte query di aggregazione. Lo sketching crea un partizionamento verticale dei dati, ovvero se consideriamo le tuple rappresentabili come vettori allora lo sketches diminuisce la dimensione di questi vettori. Formalmente, uno sketch di un vettore può essere rappresentato in questo modo:

  $$S(t)[i] = t \cdot v_i = \sum_j t[j] \cdot v_i[j]$$

  dove $S(t)$ è il vettore di sketch di dimensione k del vettore $t[1...l]$ e $v_i[1...l]$ è un vettore random ricavato da una distribuzione normale $N(0, 1)$.
  Questi vettori hanno generalmente delle proprietà che possono risultare molto utili, ad esempio la distanza tra due vettori di sketch è molto simile (errore $\leq \epsilon$) alla distaza tra i vettori dai quali sono stati ricavati. Questo aspetto presenta notevoli vantaggi in quanto permette di eseguire calcoli su vettori di dimensione minore.

- *Wavelet transoforms*: Questi metodi provengono dal mondo dell'elettronica e puntano a decomporre il segnale principale in segnali più semplici i quali hanno pochi coefficienti e possono essere usati per approssimare il segnale originale. Il problema consiste nel trovare la migliore approssimazione.

Data Stream Mining

Il data stream mining è il processo di estrazione di conoscenza da un flusso di dati continuo. Sfrutta conoscenze che derivano da diversi mondi tra cui il *data mining*, il *machine learning* e la *knowledge discovery*. In particolare, il data stream mining deve permettere agli operatori di poter essere aggiornati continuamente senza bisogno di elaborare più volte gli stessi dati (incremental evaluation). Recenti scoperte nel campo degli algoritmi di approssimazione permettono di applicare sui data stream diverse tecniche:

- *Clustering*: Il clustering viene ottenuto utilizzando la tecnica dei K-mediani (tecnica che permette di suddividere gruppi di oggetti in K partizioni sulla base dei loro attributi). I migliori algoritmi proposti utilizzano una singola passata sui dati e utilizzano poco spazio di memoria. In particolare, richiedono $O(nk)$ tempo e $O(n\epsilon)$ spazio, dove k è il numero di mediani

scelti, n è il numero di elementi totali e $\epsilon < 1$. E' stato dimostrato che nessun algoritmo che sfrutta i k-mediani può avere prestazioni migliori di $O(nk)$.

- *Classificazione*: Le tecniche di classificazione per i data stream tengono conto del fatto che i dati possono cambiare nel tempo e quindi il numero totali di classi non deve essere statico ma dinamico. Si assume che le tuple siano rappresentabili come vettori in uno spazio n-dimensionale (dove n è dato dal numero di attributi della tupla) e le tuple entranti vengono quindi proiettate in questo spazio. Quando il numero di outliers (tuple che non stanno in nessuna classe) diventano tanti e sono molto coesi tra di loro (la loro distanza è bassa) allora può essere creata una nuova classe composta da questi punti. Viceversa, quando certe tuple vengono rimosse perchè troppo vecchie certe classi possono svuotarsi molto e quindi venire rimosse.

- *Conteggio delle frequenze*: In molte query di aggregazione si è interessati solamente agli elementi che superano una certa frequenza, in quanto gli elementi con bassa frequenza sono generalmente di scarso interesse. Ad esempio, query del tipo:

  SELECT R.EventType, COUNT(*)
  FROM R
  GROUP BY R.EventType

  può essere sostituita con:

  SELECT R.EventType, COUNT(*)
  FROM R
  GROUP BY R.EventType HAVING COUNT(*)>S |R|

  dove s è una certa percentuale ed |R| è la lunghezza dello stream visto fino a quel momento.
  Gli algoritmi devono effettuare una sola passata sui dati per il calcolo di queste frequenze (incremental evaluation) mantenendo l'errore al di sotto di una certa soglia.

- *Analisi delle serie temporali*: Le serie temporali vengono usate per valutare l'andamento di un certo valore nel tempo e sono utili per identificare trend o effettuare previsioni future. Le serie temporali possono essere deterministiche, se i valori della variabile possono essere esattamente determinati sulla base dei valori precedenti, o stocastiche, se i valori della variabili possono essere determinati sulla base dei valori precedenti solo in misura parziale. Data la natura delle applicazioni di streaming, La maggioranza delle serie storiche è di tipo stocastico e si rivela quindi impossibile elaborare previsioni prive di errore.

Sliding Window

L'obbiettivo che sta dietro all'utilizzo delle sliding window è che l'utente è maggiormente interessato ad un analisi dettagliata dei dati recenti mentre sui dati storici possono essere utilizzate delle informazioni approssimative. In queste finestre ogni volta che si presenta un nuovo elemento la finestra si deve spostare in avanti e gli elementi che non rientrano più nella finestra devono essere eliminati o migrati su memoria secondaria. La finestra può essere specificata in due modi: *sequence-based window* di dimensione n, che prevede di immagazzinare all'interno della finestra gli ultimi n elementi ricevuti, e la *timestamp-based window*, che permette di immagazzinare all'interno della finestra tutte le tuple comprese nell'intervallo $[\tau - t, \tau]$, dove τ è il tempo corrente. Valutare delle query su una finestra scorrevole di dimensione n richiede $\Omega(N)$ spazio. Per computare in maniera più efficiente gli elementi all'interno della finestra ci sono diverse tecniche. Il *Windowed Sampling* prevede di trovare k campioni scelti in maniera pseudo-random per quella finestra ed eseguire le operazioni su quel campione.

Un altra soluzione potrebbe essere quella di dividere la finestra in tante piccole finestre (*subwindow* o *basic window*) e memorizzare dei synopses (riassunti), con il relativo timestamp, per ciascuna subwindow. Quando la finestra si sposta in avanti i synopses vecchi vengono eliminati e nuovi synopses vengono calcolati. Questi metodi approssimativi permettono di eseguire in maniera più rapida e con minor consumo di memoria alcuni compiti, come la valutazione della correlazione tra diversi stream o calcolare la frequenza di apparizione di alcuni elementi. Non tutti i calcoli statistici possono essere incrementalmente rivalutati, quindi l'uso dei synopses non è sempre la soluzione più adeguata.

3.3.4 Ottimizzazioni e performance delle query continue

In questa sezione si discuteranno i problemi legati al query processing e all'ottimizzazione delle query continue. Inoltre si dettaglieranno le tecniche di adaptivity e query rewriting e si forniranno alcune metriche per valutare le prestazioni e l'efficenza delle query continue.

Metriche di costo e statistiche

Le tradizionali metriche per valutare le query classiche dei DBMS non possono essere applicate alle query continue su stream infiniti, dove il costo deve essere principalmente valutato sulla base della capacità di elaborazione per unità di tempo. Di seguito verranno elencate le metriche di costo per le streaming query e le statistiche che devono essere mantenute e aggiornate:

- *Precisione e tempi di risposta vs utilizzo di memoria*: In genere, se si vuole ottenere una grande accuratezza e tempi di risposta bassi è necessario utilizzare una grande quantità di memoria primaria per cercare di mantenere quanti più dati possibili. Tecniche come il sampling o il load shedding possono essere usate per diminuire l'utilizzo di memoria ma aumentano il tasso di errore. Conoscere questi parametri è necessario per descrivere la precisione di ciascun operatore della query in funzione della memoria disponibile in modo da combinare le funzioni dei vari operatori del query plan e ottenere la precisione globale della query.

- *Output rate*: L'output rate misura il numero di tuple che escono dal sistema. E' espresso in percentuale come rapporto tra il numero di tuple entranti e il numero di tuple uscenti. Si può anche eventualmente misurare, data una certa tupla in input, il tempo necessario per produrre un output. Se l'output rate è conosciuto allora si possono applicare delle ottimizzazioni e valutare se il tasso di uscita è migliorato oppure se si sono abbassati i tempi di risposta.

- *Power usage*: Questo parametro è di fondamentale importanza delle wireless sensor network dove le batterie dei sensori hanno tipicamente una durata limitata. In questi scenari il consumo di energia deve essere minimizzato e quindi tenere traccia dei consumi (in ricezione o trasmissione) può risultare molto utile.

- *Caratteristiche dello stream*: Questa classe di parametri fornisce indicazioni sulla qualità e quantità dello stream in input, come ad esempio il tempo di inter-arrivo delle tuple, il numero di tuple perse o danneggiate e la granularità con cui le informazioni vengono ricevute. Queste informazioni possono essere utili per capire se il carico in arrivo è sostenibile dal sistema e se i dati in arrivo sono disturbati oppure no.

Query Rewriting

Il query rewriting, ovvero la riscrittura delle query, è uno dei compiti del query optimizer per cercare di ottimizzare la query. La tecnica di query rewriting cerca di riscrivere la query con un altra query semanticamente identica ma che restituisca i dati in un tempo più rapido o che li restituisca nello stesso tempo ma utilizzando meno memoria per mantenere i risultati intermedi. Ad esempio, si può riordinare la sequenza con cui vengono confrontate le clausole di una `JOIN` in modo da diminuire il numero di confronti totali da eseguire e migliorando di conseguenza i tempi di risposta. Dato che lo scopo di ogni ottimizzazione è quello di ridurre al minimo il numero di tuple estratte ed il tempo di estrazione, l'ottimizzazione definita *cost-based* (ovvero che cerca di ottimizzare qualche tipo di costo) utilizza le informazioni sulla struttura e distribuzione dei dati per assegnare un 'costo' stimato, in termini di tempo e numero di record da estrarre da ogni tabella o stream, numero di accessi, ecc. per ogni operazione. Valutando la somma totale di questi costi è possibile selezionare la sequenza più efficiente di estrazione dei dati.
Ovviamente, i costi assegnati saranno più o meno validi a seconda delle informazioni che il sistema ha e mantiene sui dati. In genere conviene mantenere le informazioni aggiornate sulla finestra temporale attuale ed eseguire il rewriting delle query su di essa, mentre per le informazioni storiche possono essere mantenute informazioni approssimate in quanto sui dati storici non vengono generalmente applicate query hard real-time.

Adaptivity

L'adaptivity è la capacità delle query di ottimizzarsi dinamicamente a seconda dei cambiamenti del flusso di dati sottostante. Il costo totale di un query plan può variare per varie ragioni: cambiamento del tempo di esecuzione di un operatore,

cambiamento della selettività di un predicato, cambiamenti dello stream, ecc. I normali query optimizer dei DBMS creano dei query plan statici in quanto le query sono tipicamente one-shot, ma per quanto riguarda le query continue su flussi di dati che variano dinamicamente nel tempo una soluzione di questo tipo può risultare poco efficiente. Alcuni sistemi offrono come soluzione a questo problema la creazione di uno scheduler che indirizza tupla per tupla verso gli operatori del query plan. In questo modo, il query plan viene automaticamente cambiato in quando lo scheduler decide a quale operatore mandare quale tupla a seconda delle condizioni del sistema. Le politiche di routing devono valutare real-time quale sono gli operatori più rapidi e selettivi per un certo stream in modo da dargli la precedenza. Anche in questo caso, bisognerà gestire il trade-off tra capacità di adattamento e l'overhead causato dalla gestione tupla per tupla dello scheduler.

Capitolo 4

Prototipo software sviluppato

Nel capitolo precedenti sono stati analizzati in dettaglio gli aspetti architetturali e funzionali di un Data Stream Management System che separa il modulo di Store Manager da quello di Query Engine. Particolare attenzione è stata dedicata alle caratteristiche che deve avere uno Store Manager System (SMS) per poter gestire efficientemente il flusso di dati in arrivo. Inoltre sono stati elencati una serie di parametri di configurazione e strutture dati per rendere l'SMS il più possibile flessibile, dinamico ed efficiente. Tutto questo è stato fatto cercando sempre di astrarre dall'effettiva implementazione. In questo capitolo invece si dettaglieranno gli aspetti implementativi di un prototipo DSMS: verrà fornita una descrizione generale del sistema e verranno descritti i requisiti funzionali e gli use case diagram dell'applicazione. Infine, verranno dettagliate le classi da cui è composta l'applicazione, gli algoritmi e l'interfaccia grafica.

4.1 Scelte tecnologiche e programmative effettuate

Al giorno d'oggi sono disponibili sul mercato una grandissima quantità di ambienti di sviluppo e linguaggi di programmazione con cui poter lavorare e bisogna essere in grado, in prima battuta, di scegliere gli strumenti più adatti alle proprie esigenze e al proprio obiettivo. Inizialmente si era pensato di estendere il database PostgreSQL aggiugendo delle funzionalità per gestire efficientemente degli stream di dati. Tale soluzione è stata immediatamente scartata in quanto modificare il codice sorgente di PostgreSQL avrebbe richiesto molto più tempo di

quello messo a disposizione dalla Tesi. Successivamente, anche prendendo spunto da altri sistemi già esistenti, si è scelto di implementare il DSMS utilizzando la tecnologia JAVA con il supporto del DBMS PostgreSQL a cui è stato installato l'estensione spaziale PostGIS per poter gestire le informazioni georefenziate. Questa scelta è stata fatta in quanto JAVA, oltre che permettere una notevole velocità nella stesura dei programma, possiede un driver di connessione verso il database PostgreSQL molto rapido (JDBC di tipo 4, che converte le chiamate ai metodi JDBC nel protocollo nativo del database). Tale caratteristica è molto importante in quanto il prototipo sviluppato dovrà gestire una grossa mole di dati e poter interrogare o recuperare dati dal database velocemente è un requisito obbligatorio. Il database PostgreSQL è stato scelto in quanto fornisce l'utilissima estensione PostGIS che permette di gestire dati georeferenziati e geometrici in maniera molto efficiente, mettendo a disposizione funzionalità ad alto livello per poter interrogare tali dati, come il calcolo della distanza tra punti, l'intersezione e l'inclusione tra aree, le lunghezze di certe linee (strade) e tanto altro. Queste funzionalità risultano fondamentali in un contesto di gestione di dati provenienti da veicoli urbani in cui le funzionalità di ricerca del percorso ottimo, di rilevamento di incidenti e di controllo sulla viabilità richiedono di gestire dati geografici.

Come ambiente di sviluppo si è preferito usare NetBeans della Sun Microsystems in quanto presenta numerosi plug-in che facilitano la stesura del programma e dell'interfaccia grafica.

4.2 Descrizione generale del prototipo

Il prototipo implementa un DSMS in cui la parte di query engine è ben separata dalla parte di gestione dei dati.

Il primo aspetto da considerare è il formato dei dati che il sistema dovrà leggere. Lo stream in ingresso è stato simulato creando diversi file in cui su ciascuna riga è riportata una tupla che indica le informazioni trasmesse dalla OBU. Il record avrà questa struttura:

<CARID, SPEED, LAT, LONG>

`CARID` rappresenta l'identificativo della vettura, `SPEED` è la sua velocità e `LAT` e `LONG` sono rispettivamente la latitudine e longitudine del veicolo.

Ad intervalli regolari (configurabili dall'utente), il sistema legge i file ed inserisce le tuple all'interno delle strutture dati, aggiungendo a ciascuna un campo `TIME`, che identifica il tempo in secondi di inserimento della tupla all'interno del sistema e prendendo come tempo iniziale l'istante in cui è stato fatto partire il programma, e `TIMESTAMP`, che è un intero crescente che identifica le tuple appartenenti ad uno stesso file.

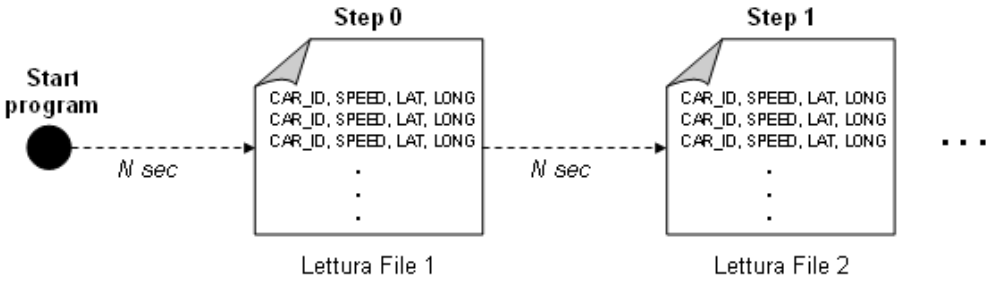


Figura 4.1: Simulazione dello stream proveniente dalle OBU.

Le tuple vengono mantenute in strutture dati che possono variare dinamicamente a seconda delle query che vengono sottomesse nel sistema. Le strutture dati, i parametri di configurazione e le politiche che decidono lo switch da una struttura ad un altra sono già state descritte nella sezione 3.2 del capitolo 3.
L'utente può sottomettere uno o più query continue o ad hoc, ciascuna con una propria finestra temporale, un tempo di riesecuzione (se continua) e dei propri vincoli sulle tuple che vengono ritornate (es. SPEED > x, LAT = y, ecc.). Il sistema automaticamente ritornerà tutte le tuple che soddisfano la query e le scriverà su un file di output. Queste tuple verranno marcate come 'consumate' e migrate sul database dal garbage collector. Le query agiranno su una certa finestra temporale e i dati che vengono mantenuti in memoria sono quelli che cadono all'interno della finestra temporale più grande. Quando tale finestra si sposta in avanti, i dati vecchi vengono migrati sul database PostgreSQL e lo spazio in memoria viene liberato.

Per permettere l'esecuzione parallela di diverse query, il prototipo è stato sviluppato seguendo un approccio *multi-thread*. Quando una query viene lanci-

ata, il Query Processing Engine crea un thread che manda le richieste allo SMS. Lo sleep time del thread è definito dall'utente e definisce il tempo di riesecuzione della query continua. Se invece la query è ad-hoc, il thread viene creato, eseguito e infine terminato. Per garantire la consistenza dei dati in un ambiente multi-thread sono stati utilizzati dei *monitor*, che in JAVA vengono implicitamente implementati tramite la keyword `synchronized`. Tale keyword permette di specificare quelle che sono le *sezioni critiche* di un oggetto, ovvero quei metodi (o anche porzioni di codice) che possono essere acceduti solo da un thread alla volta. Questo meccanismo di sincronizzazione esegue un *lock* sull'intero oggetto, in modo che se un thread sta eseguendo un metodo sincronizzato, nessun altro thread possa accedere a nessun metodo synchronized dello stesso oggetto. In questo modo si garantiscono letture e scritture consistenti. Inoltre è stata inserita una variabile esplicita di lock sull'OBU Store in modo che quando il produttore sta eseguendo il ciclo di inserimento di dati nessun consumatore possa accedere allo store. I metodi `wait()` e `notifyAll()` sull'oggetto regoleranno questo comportamento.

L'architettura del sistema può essere raffigurata nella figura 4.3. L'utente, tramite l'interfaccia, può decidere di lanciare o stoppare il DSMS: quando viene lanciato, lo Store Management System legge le tuple contenute nei file ad intervalli regolari (deciso dall'utente) e le inserisce all'interno delle strutture dati, quando viene fermato, l'SMS smette di leggere i dati dai file. Inoltre può configurare la dimensione dei blocchi referenziati dai directors (la dimensione di default è 6). Infine può sottomettere delle query consumatore in cui specifica come la query consumate i dati (Order, Eager o Never consume), la finestra temporale, il polling time e gli eventuali filtri sugli attributi. Le query vengono passate al Query Processing Engine (QPE) che si occupa di inoltrare le richieste al SMS e scrivere i risultati sui file.

Per quanto riguarda i dati migrati su DBMS, la tabella in cui vengono inseriti tali record è denominata anch'essa 'OBU store' e contiene 5 diverse colonne: `obuid`, che contiene gli identificativi delle vetture, `speed`, che contiene la velocità, `time` e `timestamp`, che rappresentano i marcatori temporali, e infine `position`, che è una colonna geometrica contenente la posizione dei veicoli approssimati come punti 2D. Il prototipo presenta tuttavia alcune limitazioni: è presente un unico storage point denominato 'OBU Store' che contiene le tuple provenienti dalle OBU e su cui verranno eseguite le query consumatore. Di conseguenza a

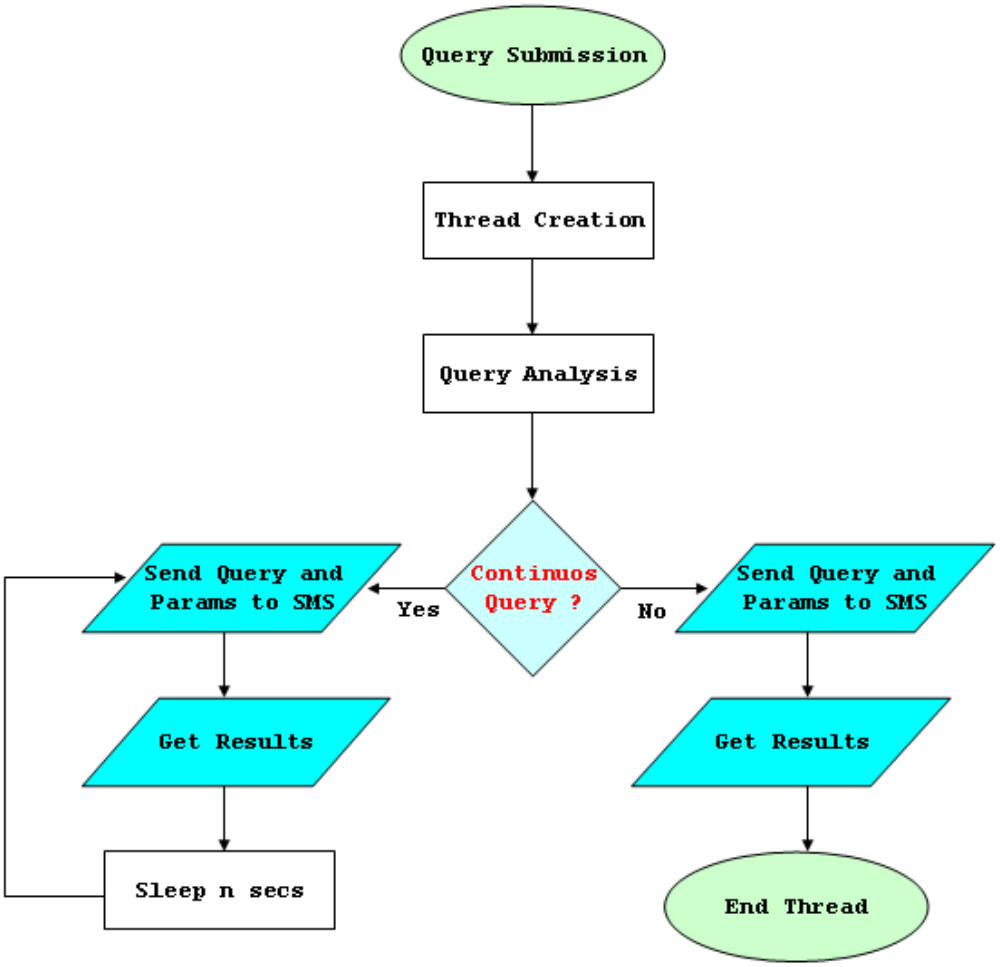


Figura 4.2: Processo di esecuzione di una query.

questa limitazione, non è possibile quindi sottomettere altre query produttore. Nel sistema ci sarà quindi un unica query produttore che si occuperà di leggere i dati dai file e inserirli nell'OBU Store. Infine, non è stato previsto alcune meccanismo di ottimizzazione delle query in quanto il focus della Tesi è stato orientato verso una gestione efficiente dei dati piuttosto che ad una ottimizzazione delle query.

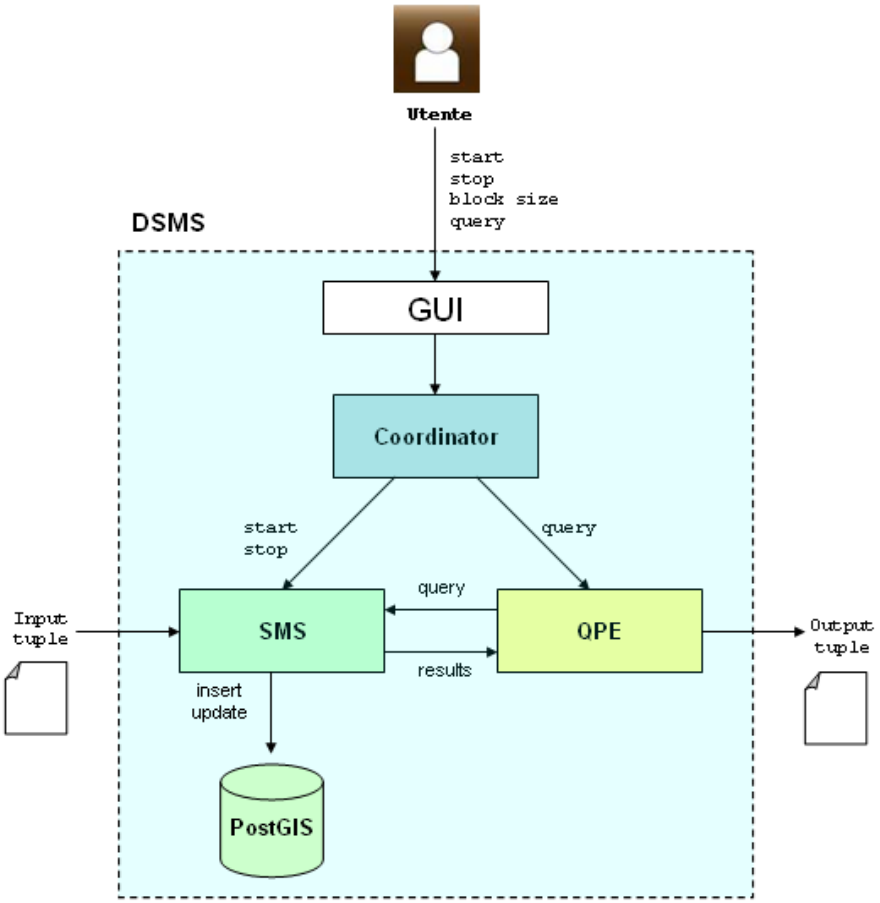


Figura 4.3: Architettura del prototipo DSMS sviluppato.

4.3 Requisiti funzionali

In questa sezione vengono elencati e dettagliati i requisiti funzionali del prototipo. Ciascun requisito ha una struttura staticamente definita che consta di: numero identificativo, macrofunzione, nome, introduzione, input, processing (descrizione processo) e output.

4.3.1 RF01: Lettura dei dati provenienti dalle OBU

- *Introduzione*: Il sistema deve essere in grado di leggere e gestire lo stream di dati in arrivo dalle OBU. Questa operazione è eseguita in automatico dal

sistema ad intervalli regolari ed è la prima operazione che viene eseguita. Questa procedura è obbligatoria e preliminare ad ogni altra operazione.

- *Input*: Per eseguire questa operazione il sistema deve ricevere alcune informazioni: file che contengono i record delle OBU, tempo di sleep del thread produttore, modalità di inserimento (never expire, order expire, ecc.) e dimensione dei blocchi puntati dai directors. I file vengono letti uno ed uno ad intervalli regolari. Il primo file a venire letto sarà `obu1.txt`, il secondo `obu2.txt`, ecc.

- *Processing*: Viene lanciato il thread produttore che, come primo passo, verifica l'esistenza del primo file da leggere (obu1.txt). Se esiste, l'OBU Store viene bloccato (lock) e si inseriscono nella struttura dati (che dipenderà, oltre che dai consumatori, anche dalla modalità di inserimento del produttore) i record del file. Quando tutti i record sono stati letti ed inseriti il file viene chiuso e il thread si mette in sleep dopo aver sblocato (unlock) l'OBU Store.

- *Output*: In caso di successo, i record risultano correttamente memorizzati all'interno della struttura dati. In caso di errore, sarà sollevata una eccezione contenente la descrizione del problema.

4.3.2 RF02: Adattamento dinamico delle strutture dati

- *Introduzione*: Il sistema deve essere in grado di cambiare dinamicamente le strutture dati in base alla modalità di inserimento del produttore e alle modalità di lettura da parte dei consumatori. La tabella che descrive questo comportamento è stata riportata nel precedente capitolo in figura 3.5.

- *Input*: Informazioni sul produttore (Never Expire, Order Expire, ecc) e sui consumatori (Never Consume, Order Consume, ecc).

- *Processing*: Quando lo SMS viene fatto partire, la struttura dati viene inizialmente creata sulla base del produttore in quanto non sono presenti consumatori. Quando un consumatore viene sottomesso, lo SMS valuta, in base al produttore e ai consumatori presenti, quale struttura dati conviene

utilizzare. Nel caso ci sia bisogno di passare da una struttura ad un altra, si esegue il metodo di switching.

- *Output*: Se tutto va a buon fine si esegue, se necessario, il cambiamento della struttura dati. In caso di errore, sarà sollevata una eccezione contenente la descrizione del problema.

4.3.3 RF03: Gestione di più finestre temporali su dati condivisi

- *Introduzione*: Nel sistema potranno essere contemporaneamente in esecuzione diverse query. Queste query accederanno allo stesso set di dati e potranno avere finestre temporali diverse, pertanto, oltre che prevedere meccanismi di sincronizzazione in modo da garantire letture consistenti da parte dei consumatori, bisognerà tenere traccia di quale consumatore ha consumato quale dato, in modo che se un record viene consumato da tutti i consumatori che cadono all'interno di una certa finestra temporale possa essere migrato su database, e che un consumatore non possa accedere più volte allo stesso dato.
  Forniamo un esempio per spiegare meglio questo requisito di grande importanza e che rappresenta una parte cruciale del prototipo: supponiamo di avere 3 query consumatore con finestre rispettivamente di 20, 15 e 10 secondi e polling time pari alla grandezza della rispettiva finestra temporale (in modo da evitare che un consumatore possa accedere più volte alla stessa tupla). Come mostrato in figura 5.1, abbiamo che se la tupla t1 viene consumata dal consumatore C3, allora può essere marcata come 'consumata' nonostante sono presenti altri 2 consumatori. Questo avviene perchè gli altri 2 consumatori agiscono su finestre temporali diverse e le diverse finestre non si overlappano su t1. Se invece t2 viene consumata solo da C3, allora non si migrerà su DB perchè anche C2 potrebbe accedere a tale tupla. Le tuple t3, t4 e t5 richiedono di essere accedute da tutti e tre i consumatori prima di poter essere marcate come 'consumate' e migrate su DB. Inoltre bisognerà garantire che nessuna tupla possa essere acceduta più volte da uno stesso consumatore. Considerando l'esempio precedente, supponiamo di avere un polling time di 5 secondi. Se C3 al primo passo

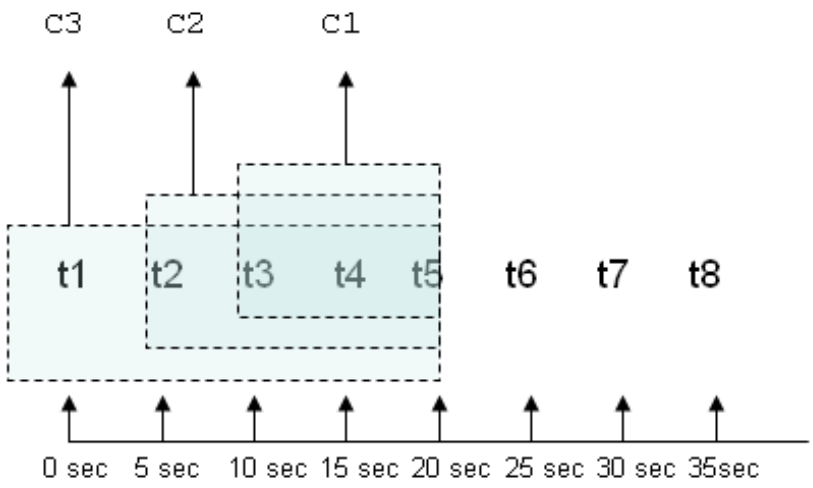


Figura 4.4: Tre diversi consumatori che agiscono su tre finestre temporali diverse.

accede a tutte le tuple, al passo dopo la finestra temporale si sposterà in avanti di 5 secondi e coprirà quindi le tuple t2, t3, t4, t5 e t6. Bisognerà fare in modo che il consumatore non possa accedere nuovamente a t2, t3, t4 e t5, ma solo a t6. Questo comportamento non è sempre richiesto, in quanto ad esempio si potrebbe desiderare che un consumatore possa accedere più volte ai dati in una finestra, ad esempio con una query del tipo: calcola la velocità media dei veicoli negli ultimi 30 secondi, con una frequenza di 5 secondi. In tal caso i dati devono essere acceduti più volte all'interno della finestra se non si prevede una media cumulativa.

- *Input*: Identificativo del consumatore che consuma la tupla, finestra temporale, polling time e modalità di consumo dei dati (Never Consume, Order Consume, ecc).

- *Processing*: Per ciascuna tupla si tiene traccia della lista di consumatori che hanno acceduto a quella tupla. Se nella tupla si overlappano n finestre temporali e la tupla è stata consumata da n consumatori diversi, allora può essere marcata come 'consumata' e successivamente migrata su database. Inoltre, tenere traccia di quale consumatore ha acceduto a quale tupla permette di evitare che un consumatore possa accedere più volte allo stesso dato. Infine, se si vuole che un consumatore possa accedere diverse volte

allo stesso dato basterà impostare in input la modalità di consumo come 'Never Consume'.

- *Output*: Se la tupla è stata consumata da tutti i consumatori allora viene marcata come 'consumata' e successivamente migrata su database da parte del garbage collector dello SMS. Viceversa, la tupla viene mantenuta in memoria in attesa di essere consumata da altri consumatori fino allo scadere della finestra temporale.

4.3.4 RF04: Migrazione dei dati storici o consumati in memoria secondaria

- *Introduzione*: Il sistema deve migrare su database PostgreSQL i dati che diventano storici o quelli consumati, in quanto non più utili per le applicazioni real-time.
- *Input*: Tupla storica o consumata.
- *Processing*: Una tupla diventa storica quando la finestra temporale più grande si sposta in avanti e non copre più tale tupla. Una tupla diventa consumata quando, come spiegato nel RF03, è stata acceduta da tutti i consumatori la cui finestra temporale racchiude tale tupla. Lo SMS, durante il processo di garbage collection, aprirà una connessione JDBC verso il DB PostgreSQL e inserirà all'interno della tabella 'OBU Store' le tuple storiche o consumate.
- *Output*: Le tuple vengono migrate su DB e lo spazio in memoria viene liberato.

4.3.5 RF05: Esecuzione parallela delle query e restituzione dei risultati

- *Introduzione*: Nel sistema potranno essere in esecuzione contemporaneamente diverse query continue o ad hoc. Il sistema deve garantire letture e scritture consistenti tramite l'utilizzo di metodi sincronizzati sulle sezioni

critiche.I risultati delle letture verranno scritti in output all'interno di file di testo e ciascun consumatore avrà un file diverso.

- *Input*: Informazioni sui consumatori: identificativo, finestra temporale, polling time e filtri.
- *Processing*: Quando un consumatore viene sottomesso viene creato un thread nel QPE che esegue le richieste allo SMS. Se la query è continua, le richieste saranno effettuate ad intervalli regolari. Viceversa, per le query one shot il thread effettua la richiesta, scrive i risultati e poi termina. All'interno dello SMS i metodi di lettura saranno sincronizzati in modo da bloccare l'oggetto che contiene i dati ed evitare delle scritture durante il processo di lettura.
- *Output*: File contenenti i risultati delle letture.

4.4 Use Case Diagram

In questa sezione analizzeremo i diagrammi dei casi d'uso, ovvero forniremo una descrizione completa delle funzioni e servizi offerti dal sistema così come sono percepiti e utilizzati dagli attori che interagiranno col sistema stesso.

4.4.1 Use Case 1: Esecuzione del produttore

Lo scenario considerato è quello della sottomissione di una query produttore.

L'utente, quando vuole lanciare la query produttore, deve inserire il polling time della query, ovvero l'intervallo di tempo con il produttore legge i dati delle OBU dai file, la dimensione dei blocchi puntati dai directors della struttura dati, la modalità di inserimento, che può essere Never Expire, Ordered Expire, Unordered Expire o Replaced Expire, e infine la dimensione della finestra temporale nel qual caso non avesse scelto come modalità di inserimento la Never Expire. Una volta inseriti questi parametri, il sistema valuta la correttezza dei campi e, se non ci sono anomalie, crea il thread produttore e la struttura dati dentro cui verranno inseriti i dati delle OBU. Il thread produttore legge i dati da file e li

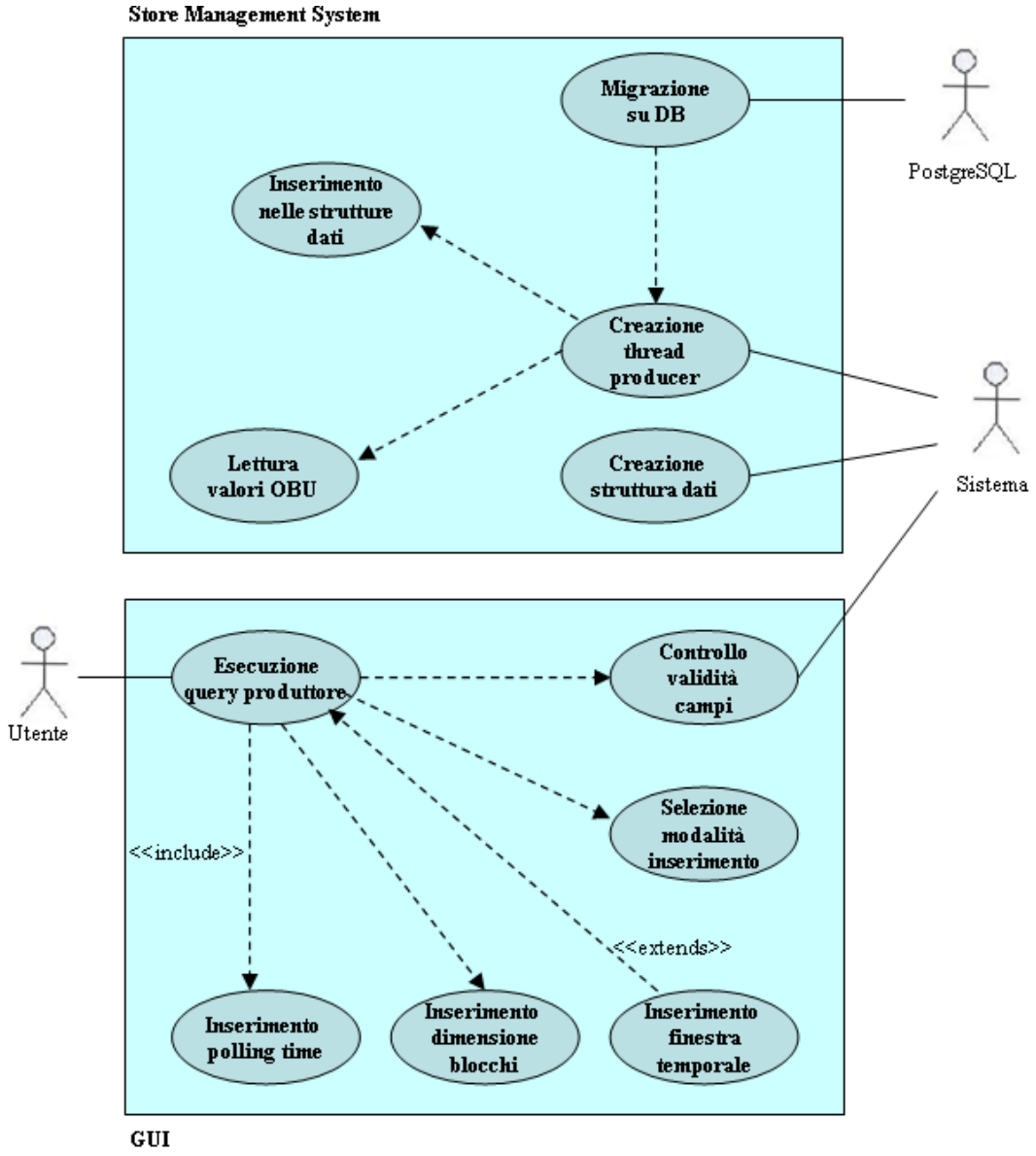


Figura 4.5: Use Case 1. Esecuzione della query produttore da parte di un utente.

inserisce all'interno della struttura dati. Infine, se una tupla diventa vecchia o consumata, il thread può migrarla sul database PostgreSQL.

Rappresentiamo tale requisito in maniera dettagliata:

- Obbietivo: Permettere l'inserimento di una query produttore.
- Pre Condizioni: I dati devono essere inseriti correttamente.

- Attore Primario: Utente.
- Attore Secondario: Sistema.
- Evento Scatenante: Richiesta di esecuzione di una query produttore.
- Sequenza:
  1. L'utente seleziona la modalità di inserimento dei dati.
  2. L'utente inserisce la dimensione dei blocchi.
  3. L'utente inserisce il polling time.
  4. Se l'utente non ha selezionato la modalità Never Expire:
     (a) L'utente inserisce la finestra temporale.
  5. L'utente seleziona l'esecuzione della query produttore.
  6. Se i dati vengono inseriti correttamente:
     (a) Il sistema crea il thread con i parametri specificati.
     (b) Il sistema crea la struttura dati con i parametri specificati.
  7. Se i dati non sono stati inseriti correttamente:
     (a) Il sistema notifica un messaggio di errore all'utente con la descrizione del problema.
- Post Condizioni (successo): Il thread produttore viene eseguito e i dati vengono letti dai file.
- Post Condizioni (fallimento): Viene notificato un messaggio di errore e il thread non viene eseguito.

4.4.2 Use Case 2: Sottomissione di una query consumatore

Lo scenario considerato è quello della sottomissione di una query consumatore.

Per inserire una query consumatore l'utente deve specificare vari parametri:

- Modalità con cui il consumatore consuma i dati (Never Consume, Order Consume o Eager Consume).

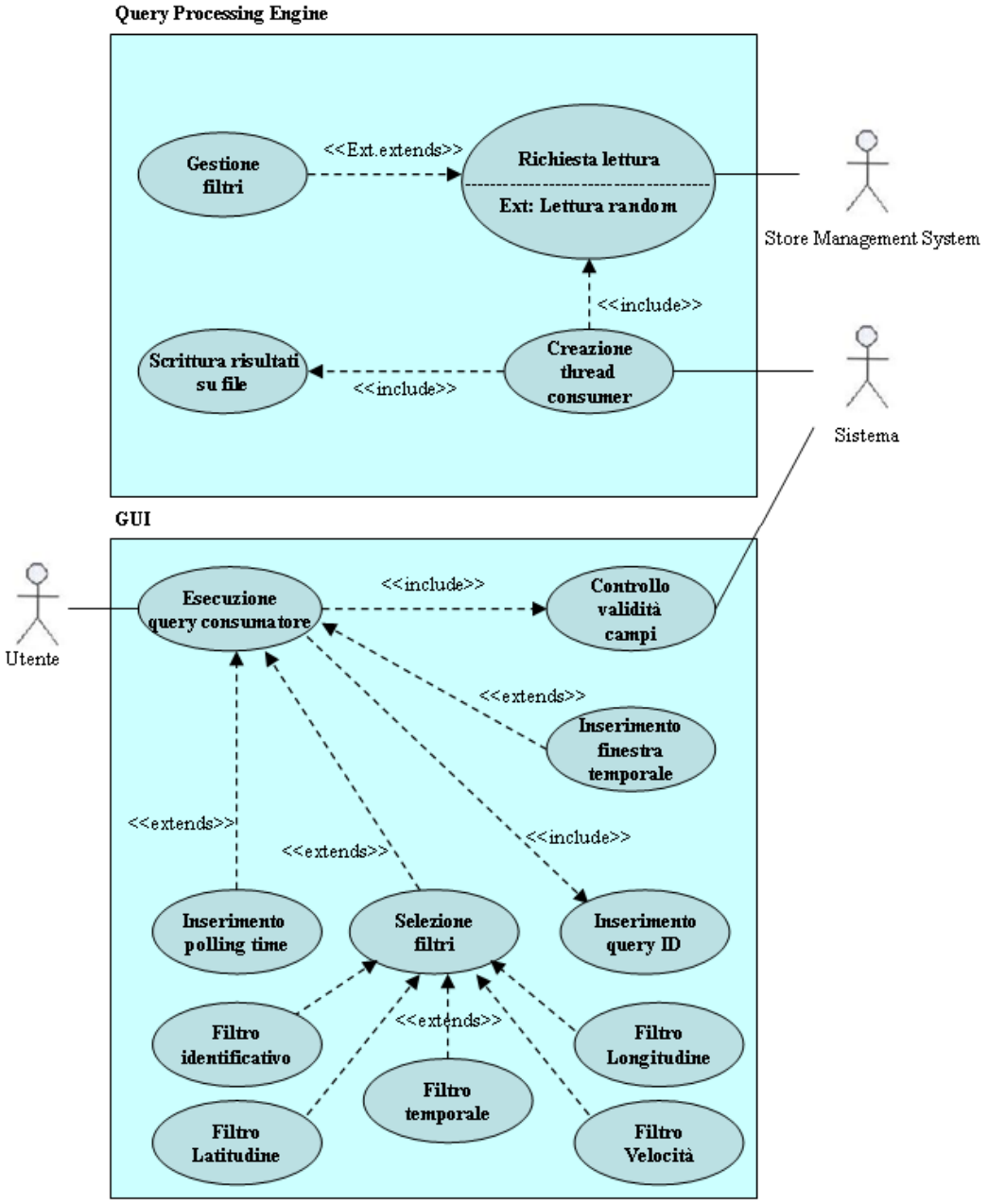


Figura 4.6: Use Case 2. Sottomissione di un query consumatore da parte di un utente.

- Il polling time (nel caso di query continua).
- La finestra temporale (nel caso di query non Never Consume).
- L'identificativo della query, che deve essere univoco all'interno del sistema.

- Gli eventuali filtri per la lettura.

I filtri sono espresse tramite operatori di confronto rispetto certi attributi. Ad esempio, è possibile richiedere di restituire solo le tuple la cui latitudine è pari ad un certo valore. Le clausole di confronto possono essere: $<$, $\leq$, $==$, $\geq$ e $>$. Gli attributi con cui è possibile effettuare dei confronti sono: identificativo della vettura, velocità, latitudine, longitudine e tempo.

Rappresentiamo tale requisito in maniera dettagliata:

- Obbietivo: Permettere l'inserimento di una query consumatore.
- Pre Condizioni: La query produttore deve essere già in esecuzione.
- Attore Primario: Utente.
- Attore Secondario: Store Management System.
- Evento Scatenante: Richiesta di esecuzione di una query consumatore.
- Sequenza:
  1. L'utente seleziona la modalità di consumo dei dati.
  2. L'utente inserisce l'identificativo della query.
  3. Se l'utente vuole eseguire una query continua:
     (a) L'utente inserisce il polling time.
  4. Se l'utente non ha selezionato la modalità Never Consumo:
     (a) L'utente inserisce la finestra temporale.
  5. Se l'utente vuole sottomettere dei filtri nelle letture
     (a) L'utente seleziona i filtri desiderati.
  6. L'utente seleziona l'esecuzione della query consumatore.
  7. Se i dati vengono inseriti correttamente:
     (a) Il sistema crea il thread consumatore con i parametri specificati.
  8. Se i dati non sono stati inseriti correttamente:
     (a) Il sistema notifica un messaggio di errore all'utente con la descrizione del problema.

- Post Condizioni (successo): Il thread consumatore viene eseguito ed effettua le richieste al SMS. I risultati vengono scritti sul file di output relativo a quella query consumatore. Se la query è one shot allora il thread viene terminato dopo aver effettuato una richiesta.
- Post Condizioni (fallimento): Viene notificato un messaggio di errore e il thread consumatore non viene eseguito.

4.5 Moduli e macro funzionalità

Il prototipo sviluppato è fortemente modulare e quindi implementato su diverse classi che svolgono compiti ben distinti. In questa sezione verranno descritte le classi da cui è composto il programma e le relative funzionalità.

4.5.1 Grafo delle classi

Analizziamo il grafo delle classi:

Dalla figura 4.7 si nota che la classe DSMSCoordinator gestisce il funzionamento dell'applicazione. Contiene il `main()` e si occupa di far partire l'interfaccia grafica, lo storage manager e sottomette le query consumer al query engine. L'utente interagisce con l'interfaccia grafica per lanciare i comandi di sottomissione della query producer (e quindi di esecuzione dello storage manager) e delle query consumer e riceve le informazioni notificate dal sistema. Il query engine, tramite l'interfaccia messa a disposizione, invia i comandi di lettura allo storage manager il quale restituisce le tuple che soddisfano le condizioni di lettura all'interno di `resultSet`. Le classi `DSMSInterface`, `StorageManager` e `QueryEngine` utilizzano le classi `DataStructure`, `Consumer` e `Tuple` che contengono la definizione degli oggetti consumatore, tupla e la definizione degli schemi. Queste classi verranno analizzate meglio in seguito.

4.5.2 Classe DSMSCoordinator.java

Questa classe svolge il ruolo di coordinatore del DSMS. Contiene il `main()`, il quale si occupa di far partire l'interfaccia grafica. Tramite l'interfaccia grafica

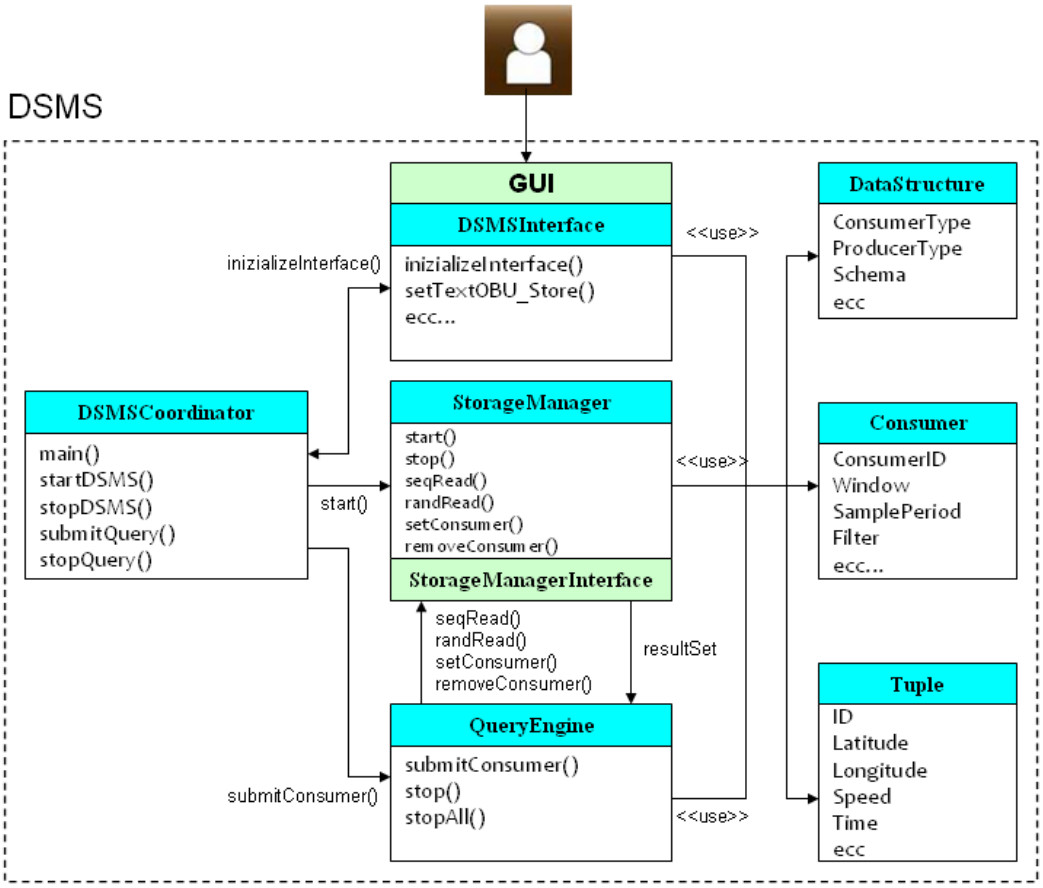


Figura 4.7: Grafo delle classi.

l'utente comunicherà al DSMS la tipologia di query produttore e consumatore da sottomettere al sistema e questa classe si occuperà di inviare le richieste allo store manager e al query engine. In particolare, i seguenti metodi statici saranno richiamati dalla classe che implementa l'interfaccia grafica:

- `startDSMS`: Data la dimensione dei blocchi, il tipo di produttore, il polling time e la finestra temporale viene comunicato allo storage manager di eseguire la query produttore. Quando viene lanciato questo metodo il DSMSCoordinator crea un istanza dell'oggetto `StorageManager` (che implementa lo Store Manager System) e invoca il metodo `start()` passandogli i parametri specificati. Inoltre viene anche creata un istanza dell'oggetto `QueryEngine` (che implementa il Query Processing Engine) a cui viene passato il riferimento all'oggetto `StorageManager`, in modo da dargli la

possibilità di inoltrare successivamente le richieste di lettura. Il query engine inizialmente non eseguirà nessuna query consumatore e rimarrà quindi in attesa.

- `submitQuery`: Dato un particolare consumatore, si occupa di comunicare al query engine la creazione di una query consumatore. Il consumatore conterrà tutte le informazioni che descrivono una query consumatore (ID, finestra temporale, ecc.). Quando viene chiamato questo metodo il DSMSCoordinator invoca la funzione `submitConsumer` sull'oggetto QueryEngine precedentemente creato e gli passa l'oggetto `Consumer` che contiene la descrizione della query consumatore.

- `stopDSMS`: Ferma lo storage manager e il query engine. Vengono quindi richiamati sull'oggetto `StorageManager` e `QueryEngine` rispettivamente i metodi `stop()`, che ferma la query produttore, e `stopAll()`, che ferma tutte le query consumatore presenti nel query engine.

- `stopQuery`: Dato un particolare query ID, comunica al query engine di fermare una particolare query consumatore. Questo metodo richiama il metodo `stop()` sull'oggetto `QueryEngine` passandogli l'identificativo della query consumatore da fermare.

In pratica, il DSMSCoordinator svolge un ruolo di *middleware* tra l'interfaccia grafica, o eventualmente altre applicazioni, e il funzionamento del Data Stream Management System. L'utente o lo sviluppatore dovranno semplicemente richiamare questi metodi per gestire il funzionamento del DSMS. Si è voluto aggiungere questo strato aggiuntivo in quanto aumenta la sicurezza e la semplicità d'uso del sistema.

4.5.3 Classe StorageManager.java

L'implementazione dello store manager è racchiusa in questa classe. Come primo passo per lo sviluppo di questo modulo è stata definita l'interfaccia astratta che definisce i metodi che sarebbero dovuti essere stati implementati. Tale interfaccia è definita nel file `StorageManagerInterface.java` e contiene diversi metodi tra cui:

- `start()`: Questo metodo esegue la query produttore. Viene eseguito un thread che ad intervalli regolari legge i file che contengono le segnalazioni delle OBU. Viene richiamato dal `DSMSCoordinator`.

- `stop()`: Ferma il thread produttore. Viene richiamato dal `DSMSCoordinator`.

- `setConsumer()`: Questo metodo viene richiamato dal query engine e comunica allo store manager la tipologia di consumatore da inserire. In particolare, lo store manager registra tutti i consumatori che vengono comunicati per un certo store (nel nostro caso solo l'OBU Store) e in base ad essi (e al produttore) decide quale struttura dati utilizzare. Inoltre, le informazioni sui consumatori vengono anche usate per mantenere la finestra temporale sui dati. Ad esempio, se vengono comunicati 3 consumatori con la finestra temporale di 5, 10 e 15 secondi, allora lo store manager manterrà una finestra temporale sui dati di 15 secondi in modo da poter mantenere in memoria tutti i dati di cui i consumatori necessitano. Se invece le finestre sono di 5, 10 e 12 secondi allora la finestra temporale sarà di 12 secondi.

- `removeConsumer()`: Rimuove un consumatore da un certo store. Quando uno store viene rimosso, lo store manager cambia eventualmente la struttura dati e aggiorna la finestra temporale. Ad esempio, se sono stati registrati un consumatore C1 di tipo Eager Consume con una finestra di 15 secondi e un consumatore C2 di tipo Order Consume con una finestra di 10 secondi e il produttore e Ordered Expire, se il consumatore C1 viene rimosso allora la struttura dati viene cambiata in FIFO e la finestra temporale viene cambiata da 15 a 10 secondi. Questo metodo viene sempre richiamato dal query engine.

- `sequentialRead()`: Questo metodo permette al query engine di effettuare richieste sequenziali. Al metodo viene passato l'identificativo del consumatore e vengono ritornate tutte le tuple all'interno della finestra temporale relativa a quel consumatore.

- `randomRead()`: Questo metodo permette al query engine di effettuare richieste con filtri, ovvero in cui sono presenti delle condizioni sulle tuple da restituire. Al metodo viene passato l'identificativo del consumatore

e vengono ritornate tutte le tuple all'interno della finestra temporale che soddisfano le condizioni richieste dal filtro.

Il passo successivo è stato implementare le strutture dati. E' stata definita una classe astratta `GenericStorage` che implementa i metodi comuni a tutte le strutture dati, come la registrazione e cancellazione di consumatori, la lettura di valori, le stampe, ecc. Sono invece stati definiti come astratti i metodi di inserimento di valori, in quando differenti strutture dati seguono 'filosofie' di inserimento differenti, e il metodo di migrazione dei dati su database, in quando la struttura dati IN-PLACE non fa uso di finestre temporali per migrare i dati su DB e quindi il metodo deve essere implementato diversamente rispetto alla FIFO e alla RANDOM. Da questa classe astratta sono state quindi derivate le classi figlie FIFO, RANDOM e INPLACE che implementano rispettivamente la struttura dati FIFO, RANDOM e IN-PLACE.

Per permettere il cambiamento dinamico da una struttura dati ad un altra è stato sfruttato il *polimorfismo*. In pratica, l'OBU store è stato definito come un oggetto `GenericStorage` che, in base al produttore e ai consumatori presenti, viene specializzato nella classe FIFO, RANDOM o INPLACE. I metodi che permettono di effettuare lo *switching* tra una struttura dati ed un altra sono `switchToFIFO` e `switchToRANDOM`. Non è presente un metodo di tipo switchToIN-PLACE in quanto la struttura dati IN-PLACE non subisce mai dei cambiamenti dinamici.

L'ultimo passo è stato gestire l'esecuzione concorrente di diversi thread su dati condivisi. Per fare questo, i metodi di inserimento, lettura e migrazione di valori sono stati definiti con la keyword `synchronized` in modo che un solo thread per volta possa avere accesso esclusivo alla struttura dati. In questo modo si garantiscono letture consistenti evitando operazioni di switching o inserimento di valori durante una lettura. Inoltre, quando il thread produttore esegue il ciclo di inserimento dati (lettura file), vengono utilizzate delle variabili di `lock` sullo store in modo che nessun consumatore possa accederci finchè il ciclo di inserimento non è concluso. A questo scopo, all'interno della classe astratta `GenericStorage`, sono stati creati i metodi `lockStorage()`, `unlockStorage` e `checkIfLock()` che permettono al produttore di bloccare o rilasciare una variabile booleana di lock e ai consumatori di verificare lo stato di questa variabile. I consumatore quindi, quando dovranno eseguire delle lettura, controlleranno tramite il metodo `checkIfLock()` lo stato dello store. Se il metodo ritorna `true` significa che lo

store è bloccato e quindi eseguiranno una `wait()`, mentre se è `false` significa che è possibile effettuare la lettura. Il produttore, quando rilascia lo store, oltre che cambiare lo stato del lock tramite il metodo `unlockStorage()`, eseguirà una `notifyAll()` in modo da risvegliare tutti i thread che erano in attesa.

4.5.4 Classe QueryEngine.java

Questa classe implementa il query engine del DSMS. Riceve le query consumatore dal `DSMSCoordinator` e le esegue effettuando le richieste di lettura allo store manager. Per ogni query consumatore che viene sottomessa viene creato un thread che, in base alla presenza di filtri o meno, effettua le richieste di letture sequenziali o random allo store manager. Il query engine tiene traccia di tutti i consumatori sottomessi in un `ArrayList` di tipo `Consumer` in cui il thread che esegue il consumatore ha lo stesso ID del consumatore stesso. Per quanto riguarda la sottomissione dei comandi di lettura allo storage manager, all'interno della funzione run() della classe che implementa l'interfaccia `Runnable` (e che quindi vuole eseguire il thread consumatore) è presente il seguente ciclo (in pseudocodice):

```
run(){
    consumerID=nome del thread corrente
    while (true)
    if (consumer ha filtri)
        results=SMS.randomRead(consumerID)
    else
        results=SMS.sequentialRead(consumerID)
    scrivi risultati su file
    Thread.sleep(polling time del consumatore)
}
```

Lo store manager, tramite l'identificativo, riesce a risalire alle caratteristiche del consumatore e quindi agli eventuali filtri e alla finestra temporale. In prima battuta ci si potrebbe chiedere il motivo per cui lo store manager debba tenere traccia delle caratteristiche di ciascun consumatore, in quanto analizzando il codice sopra riportato bastarebbe comunicare direttamente ai comandi di lettura la dimensione della finestra temporale e gli eventuali filtri, senza necessità

che lo store manager abbia conoscenza delle caratteristiche di ciascun consumatore. Il problema nasce quando si vuole fare in modo che le strutture dati si adattino dinamicamente ai cambiamenti dei consumatori. Quando si cancella un consumatore, lo store manager deve essere a conoscenza delle caratteristiche dei consumatori rimanenti, in modo da poter eventualmente cambiare la dimensione della finestra temporale e la struttura dati. Quello che si poteva evitare è differenziare la lettura sequenziale da quella random, in quanto lo store manager, essendo a conoscenza delle caratteristiche di ciascun consumatore (e quindi anche dei filtri), poteva dedurre da solo se la lettura era sequenziale o random. Si è preferito però agire in questo modo in modo da dare una visione di quello che sarò il comportamento futuro dell'applicazione, in cui lo store manager non sarà a conoscenza dei filtri sul consumatore (ma solo della tipologia e della finestra temporale) e sarà il query engine, dopo aver eseguito il parsing della query SQL-like, a capire se la query presenta delle condizioni sugli attributi oppure no, e quindi a decidere se la lettura dovrà essere sequenziale o casuale (in cui verranno passate anche le condizioni da soddisfare).

4.5.5 Altre classi

Nel progetto, oltre le classi descritte precedentemente, sono presenti altre classi di supporto molto utili:

- `DSMSInterface.java`: Questa classe implementa l'interfaccia grafica del programma. Mette a disposizione dei metodi statici per permettere al query engine e allo store manager di mandare messaggi all'utente. I comandi e i parametri comunicati dall'utente vengono comunicati al `DSMSCoordinator`, il quale si occupa di passarli allo store manager o al query engine tramite le interfaccie messe a disposizione. Questa classe quindi non comunica mai direttamente con il query engine o lo store manager ma passa sempre attraverso al `DSMSCoordinator`. Maggiori dettagli sull'interfaccia grafica verranno forniti in seguito.

- `Consumer.java`: Questa classe descrive l'entità consumatore. Un consumatore è caratterizzato da un certo identificativo, una finestra temporale, un `ConsumerType` (Eager Consumer, Order Consume, ecc.), un sample period (o polling time, ovvero il tempo di riesecuzione) e un eventuale filtro.

Mette a disposizione dei metodi per modificare o ricevere questi parametri ed è utilizzato da quasi tutte le altre classi del sistema.

- `DataStructure.java`: Questa classe contiene la definizione (`enum`) della tipologia di consumatore, produttore, store, schema e operatori logici. In dettaglio:

  `public static enum` ConsumerType{NeverConsume, OrderedConsume, EagerConsume}
  `public static enum` ProducerType{NeverExpire, OrderedExpire, UnorderedExpire, ReplacedExpire}
  `public static` enum StorageType{GENERICSTORAGE, INPLACE, FIFO, RANDOM}
  `public static` enum Schema{ObuId, Speed, Lat, Long, Time, Timestamp, Consumed}
  `public static` enum LogicOperators{Min, MinEq, Mag, MagEq, Eq, NotEq}

- `Tuple.java`: Questa classe descrive una tupla. Le tuple comunicate dalle OBU sono caratterizzate da: identificativo, velocità, latitudine, longitudine, time, timestamp e il booleano consumed. Inoltre, per ogni tupla, viene mantenuto un vettore che mantiene gli identificativi dei consumatori che hanno acceduto a quella tupla. Quando tutti i consumatori presenti nel sistema hanno acceduto alla tupla, il flag 'consumed' viene impostato a `true`. Inoltre, mantenere questo vettore permette di evitare ad un consumatore di accedere più volte allo stesso dato. Per risparmiare in memoria inizialmente si era pensato di mantenere, per ogni consumatore, un puntatore all'ultimo elemento acceduto. Il problema nasce quando si devono gestire switch dinamici da una struttura dati ad un altra, in cui i puntatori devono cambiare dinamicamente rendendo la gestione molto più complessa. Si è preferito quindi, in questa versione iniziale, optare per una soluzione più semplice anche in vista del fatto che i consumatori sono generalmente pochi.

4.5.6 Interfaccia grafica

L'interfaccia grafica, come detto precedentemente, è implementata nella classe `DSMSInterface`. Per creare la GUI è stata utilizzata la libreria grafica `javax.swing` che permette una grande portabilità data dal fatto che i componenti sono creati utilizzando punti e linee, e tutte le API di sistema permettono generalmente un disegno di questo tipo. Il lato negativo, rispetto alle vecchie librerie `awt`, è una maggiore lentezza.

L'interfaccia utente è presentata in figura 4.8. Nel pannello denominato

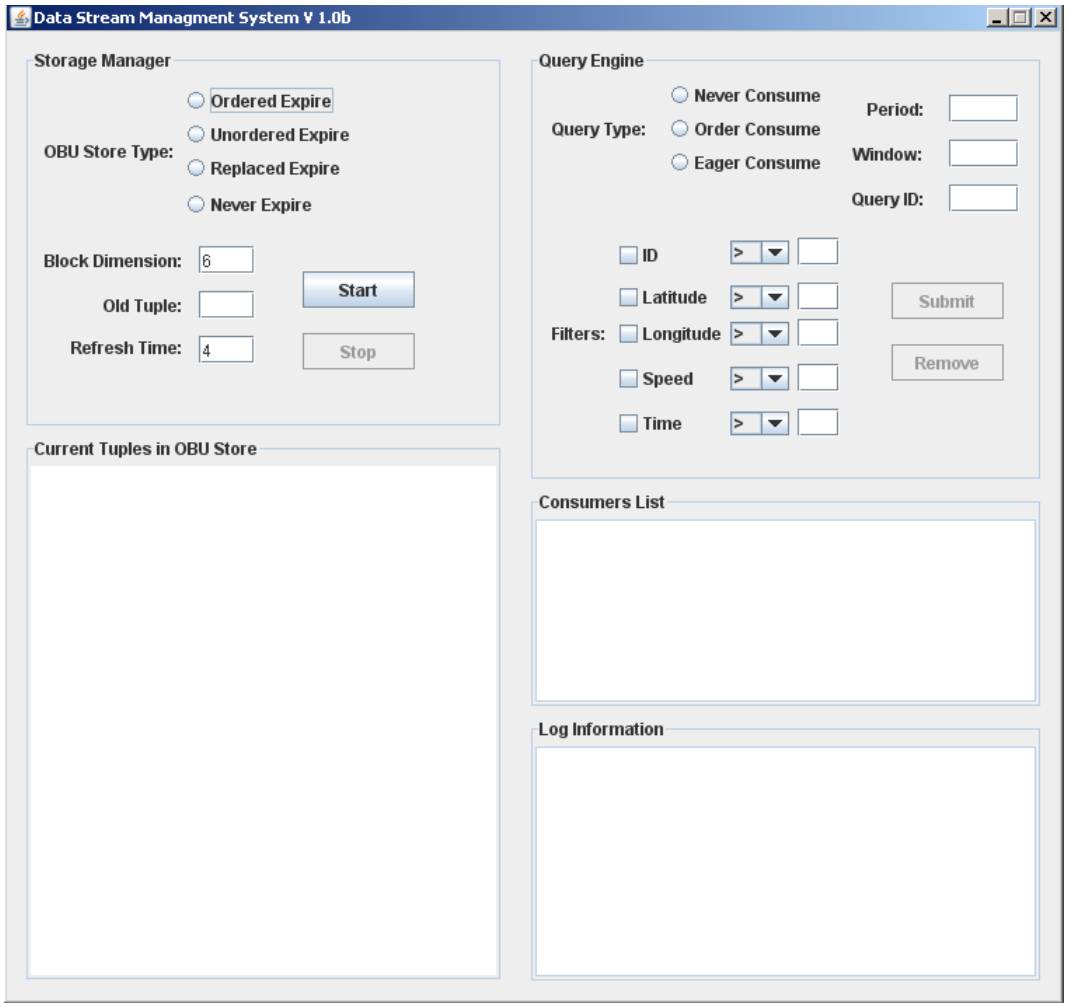


Figura 4.8: GUI utente.

`Storage Manager` si possono impostare le caratteristiche del produttore, ovvero si può selezionare: modalità di inserimento, dimensione dei blocchi, finestra tem-

porale e polling time. Tramite il pulsante di `start` si fa partire il DSMS e di conseguenza viene eseguita la query produttore, mentre tramite il pulsante di `stop` il sistema viene fermato.

Nel pannello denominato `Query Engine` vengono invece gestite le caratteristiche delle query consumatore che l'utente desidera sottomettere nel sistema. E' possibile selezionare: tipologia di consumatore, polling time, finestra temporale, identificativo della query ed eventuali filtri. Tramite il pulsante di `submit` si sottomette la query consumatore nel sistema. Prima di poter sottomettere una query consumatore deve essere in esecuzione il produttore, quindi prima si esegue lo start e successivamente si possono inserire dei consumatori tramite il submit. Inserendo nel campo `Query ID` un identificativo di una query e premendo il tasto `remove` è possibile fermare una query consumatore.

Infine, nella text area 'Current Tuples in OBU Store' vengono elencati i dati presenti nell'OBU Store, mentre in 'Consumers list' è elencata la lista di consumatori presenti nel sistema. La text area 'Log information' fornisce informazioni secondarie comunicate dal sistema.

Capitolo 5

Prove sperimentali e conclusioni

In questo capitolo vengono presentati i test effettuati sul prototipo ed i risultati ottenuti.

Come spiegato precedentemente, i vincoli legati ai sistemi per la gestione degli stream sono molto stringenti in termini di performance e scalabilità. Pertanto, nello sviluppo software del prototipo, sono state adottate tecniche programmative per rispondere velocemente alle query e gestire i dati nella maniera più efficiente possibile, cercando di ottimizzare l'utilizzo di memoria e il tempo CPU. L'obiettivo dell'analisi di performance è verificare se il sistema soddisfa certi requisiti di efficienza e può essere efficacemente utilizzato per il suo scopo.

In totale, sono stati eseguiti tre differenti test per valutare le prestazioni e la scalabilità del sistema:

- **Stress Testing**: Lo scopo principale di questo test è stato sovraccaricare di dati il sistema per verificare quante informazioni è in grado di elaborare mantenendo accettabili le performance. In questo modo si è misurata la capacità del prototipo di gestire efficientemente le informazioni senza andare in crash.

- **Load Testing**: Con questa tipologia di test si è cercato di aumentare gradualmente il numero di query consumatore misurando i tempi necessari per ottenere le risposte. Il flusso di dati in ingresso viene invece mantenuto costante.

- **Performance Testing**: Il performance testing ha avuto lo scopo di misurare le prestazioni dell'algoritmo di lettura sulle strutture dati in base

al tipologia di consumatore e ai dati presenti in memoria. Dai risultati ottenuti si è cercato di individuare una stima asintotica.

E' bene precisare che i risultati dei test effettuati sono fortemente condizionati dall'architettura hardware sulla quale sono stati eseguiti. In questo caso, è stato utilizzato un modesto portatile con processore AMD Turion 64bit 1.60GHz dual core con 896MB di RAM, di cui 256MB sempre dedicati a NetBeans. Inoltre, sopratutto per quel che riguarda lo stress testing, è stato dedicato a Java un *heap* (spazio di memoria per le allocazioni dinamiche) di 256MB. Nonostante questi vincoli stringenti si sono comunque ottenuti buoni risultati, segno che il prototipo è stato sviluppato in maniera corretta. Bisogna inoltre sottolineare che i risultati ottenuti non devono essere interpretati come informazioni estremamente accurate di quelle che saranno le prestazioni del sistema finale ma semplici indicazioni orientative, in quanto allo stato attuale al prototipo mancano troppe parti per poter effettuare delle analisi precise. In particolare, al sistema manca tutta la parte relativa al parsing ed ottimizzazione delle query utente e alle connessioni di rete verso le OBU, problematiche che vanno oltre questo progetto di Tesi ma che rappresentano un punto cruciale per le performance e la scalabilità del sistema finale.

I risultati ottenuti rappresentano comunque un interessante suggerimento per l'eventuale lavoro futuro che verrà eseguito in quest'ambito.

5.1 Dataset utilizzato

Il primo passo per la valutazione delle prestazioni è stato cercare di identificare un dataset che fosse il più possibile realistico ed indicativo. Il prototipo è stato sviluppato per leggere in polling ad intervalli regolari dei file nei quali sono contenuti i dati delle OBU. Pertanto, per quel che riguarda il dataset, si possono gestire essenzialmente 2 parametri:

- Frequenza con cui il sistema legge i file.
- Quantità di tuple all'interno dei file.

Una frequenza di lettura molto elevata, anche ipotizzando un contesto reale, permette di mantenere una 'fotografia' della realtà molto precisa in quanto i dati

inviati dalle OBU vengono campionati più spesso, ma richiede al produttore di bloccare più spesso lo store per poterci inserire i dati, peggiorando di conseguenza le prestazioni dei consumatori che devono attendere il rilascio del lock sullo store per poter leggere. Se invece si mantiene costante la frequenza ma si aumenta la quantità di tuple che il sistema riesce a leggere nell'unità di tempo si aumenta maggiormente il consumo di memoria ma se la frequenza non è molto alta i consumatori riusciranno ad accedere più spesso allo store, migliorando le performance.

Per quel che riguarda lo stress testing, il cui obbiettivo è stato 'rompere' il sistema verificando quante tuple riesce a gestire, sono stati utilizzati 20 diversi file da 10.000 tuple ciascuno in modo da verificare abbastanza accuratamente quale fosse il limite del prototipo.

Per il load testing invece si sono utilizzati 3 differenti dataset: il primo da 40 file con ciascuno 1.000 tuple, il secondo da 40 file con ciascuno 2.000 tuple e infine il terzo sempre con 40 file ma da 5.000 tuple ciascuno. Per ciascun dataset si è fatta variare la frequenza di lettura del produttore (4, 2 e 1 secondo) e, per ciascuna frequenza, il numero di thread consumatore è stato fatto variare da 1 a 128. In pratica, nel primo test si è utilizzato il primo dataset e si è impostato il polling time del produttore a 4 secondi e si sono verificati i tempi di risposta facendo variare i consumatori da 1 a 128 procedendo per potenze di 2 (1, 2, 4, 8, ecc.). Successivamente si è poi impostato il polling time a 2 secondi e sono stati rieseguiti i test. Una volta terminati tutti i test sul primo dataset si è utilizzato il secondo in modo da aumentare gradualmente la quantità di dati in arrivo al sistema. Infine è stato utilizzato il terzo dataset. In questo modo si è voluto verificare quanto la quantità di informazioni che il sistema acquisisce per unità di tempo, la frequenza di lettura del produttore e la quantità di thread consumatori presenti parallelamente nel sistema influissero sulle performance. Per le query consumatore sono state utilizzate query standard che eseguono richieste di lettura ogni 5 secondi. Questo è stato in assoluto il test eseguito con maggiore attenzione perchè misura i parametri fondamentali di un DSMS: tempi di risposta e scalabilità.

Infine, per il performance testing, è stato utilizzato come per lo stress testing un workload composto da 20 file con 10.000 tuple ciascuno. L'obbiettivo è stato misurare le prestazioni dell'algoritmo di lettura (sia random che sequenziale) all'aumentare del carico sul sistema.

5.2 Analisi delle performance e della scalabilità

In questa sezione verranno mostrati e commentati i risultati emersi dai vari test effettuati.

5.2.1 Stress Testing

Come detto precedentemente, l'obbiettivo dello stress testing è stato quello di sottoporre al sistema un carico eccessivo per misurare la quantità di informazioni che il sistema è in grado di elaborare senza problemi.

Per questo test è stato utilizzato un dataset composto da 40 file, in cui su ciascun file erano presenti 10.000 tuple. Il polling time del produttore è stato impostato a 4 secondi ed è stata eseguita anche una query consumatore per verificare le variazioni dei tempi di risposta in base alla quantità di informazioni presenti nel sistema.

Dai risultati è emerso che il sistema riesce, considerando un heap limitato di 265MB, a gestire in memoria fino a 180.000 tuple garantendo sempre tempi di risposta bassi. In generale, considerando anche una finestra temporale che copre tutti i dati, le risposte alla query vengono sempre fornite in tempi < 32ms. Superate le 180.000 tuple il sistema non riesce ad allocare nuova memoria e si blocca. Questa limitazione può essere semplicemente superata aumentando la quantità di memoria primaria a disposizione della JVM (Java Virtual Machine) oppure adottando una politica di load shedding (le tuple vengono scartate o migrate quando la memoria disponibile è esaurita). Effettuando una rapida stima, considerando uno heap di 1GB, il sistema riuscirebbe a gestire fino 720.000 tuple, ovvero riuscirebbe ad elaborare in memoria primaria le informazioni inviate da tutte le vetture presenti in città come Reggio Emilia o Modena, considerando di mantenere anche buone finestre temporali su tali dati.

5.2.2 Load Testing

Questa tipologia di test è stato di gran lunga quello più analizzato in dettaglio perchè permette di misurare bene quelle che sono le capacità del sistema in termini di performance e scalabilità. Il load testing permette di valutare i comportamenti del sistema al variare di alcuni parametri fondamentali tra cui: numero di

consumatori presenti, polling time del produttore (frequenza di campionamento) e quantità di dati in arrivo per unità di tempo.

Per eseguire questo test si è proceduti in questo modo: si è inizialmente considerato un dataset composto da 40 file da 1.000 tuple ciascuno e si sono misurati i tempi di risposta alle query variando il numero di query (thread) presenti nel sistema e la frequenza di campionamento. Al passo successivo, si è mantenuto lo stesso dataset ma si è alzata la frequenza di acquisizione dei dati (da 4 secondi a 2 e da 2 a 1 secondo) e si sono misurati nuovamente i tempi di risposta alle query. Una volta terminati tutti i test sul primo dataset, si è passati ad un dataset più consistente composto da 40 file da 2.000 tuple ciascuno e si sono eseguiti gli stessi test. Per concludere, si è considerato un dataset composto da 40 file da 5.000 tuple per file. Non si è proceduti considerando dataset più consistenti perchè il sistema non sarebbe stato in grado di garantire tempi di risposta accettabili.

Le query consumatore sottomesse nel sistema sono query continue sequenziali con un polling time di 5 secondi e una finestra temporale anch'essa di 5 secondi. Non vengono mostrati i tempi di risposta per query con filtri perchè i risultati sono pressochè equivalenti.

Analizziamo i risultati emersi di vari test:

Legenda	
TxF	Tuple per file
SP	Sampling Period (polling time produttore)
N.Thread	Numero query consumatore, 1 thread = 1 query
MRT	Max Response Time
AVG	Average. Tempo medio di risposta

TxF	**SP**	**N.Thread**	**MRT**	**AVG**
1000	4 sec	1	1 ms	1 ms
1000	4 sec	2	1 ms	1 ms
1000	4 sec	4	$\leq$ 16 ms	1 ms
1000	4 sec	8	$\leq$ 16 ms	1 ms
1000	4 sec	16	$\leq$ 94 ms	3 ms
1000	4 sec	32	$\leq$ 266 ms	8 ms
1000	4 sec	64	$\leq$ 406 ms	20 ms
1000	4 sec	128	$\leq$ 1500 ms	43 ms

1000	2 sec	1	1 ms	1 ms
1000	2 sec	2	1 ms	1 ms
1000	2 sec	4	$\leq$ 16 ms	1 ms
1000	2 sec	8	$\leq$ 16 ms	1 ms
1000	2 sec	16	$\leq$ 218 ms	7 ms
1000	2 sec	32	$\leq$ 312 ms	16 ms
1000	2 sec	64	$\leq$ 610 ms	31 ms
1000	2 sec	128	$\leq$ 1900 ms	66 ms
1000	1 sec	1	1 ms	1 ms
1000	1 sec	2	1 ms	1 ms
1000	1 sec	4	$\leq$ 16 ms	2 ms
1000	1 sec	8	$\leq$ 234 ms	24 ms
1000	1 sec	16	$\leq$ 297 ms	47 ms
1000	1 sec	32	$\leq$ 406 ms	50 ms
1000	1 sec	64	$\leq$ 594 ms	53 ms
1000	1 sec	128	$\leq$ 2297 ms	167 ms
2000	4 sec	1	1 ms	1 ms
2000	4 sec	2	1 ms	1 ms
2000	4 sec	4	$\leq$ 16 ms	2 ms
2000	4 sec	8	$\leq$ 16 ms	2 ms
2000	4 sec	16	$\leq$ 188 ms	5 ms
2000	4 sec	32	$\leq$ 328 ms	9 ms
2000	4 sec	64	$\leq$ 719 ms	30 ms
2000	4 sec	128	$\leq$ 2546 ms	57 ms
2000	2 sec	1	1 ms	1 ms
2000	2 sec	2	1 ms	1 ms
2000	2 sec	4	$\leq$ 16 ms	2 ms
2000	2 sec	8	$\leq$ 265 ms	16 ms
2000	2 sec	16	$\leq$ 438 ms	36 ms
2000	2 sec	32	$\leq$ 500 ms	47 ms
2000	2 sec	64	$\leq$ 844 ms	60 ms
2000	2 sec	128	$\leq$ 3437 ms	302 ms
2000	1 sec	1	1 ms	1 ms
2000	1 sec	2	1 ms	1 ms

2000	1 sec	4	$\leq$ 240 ms	27 ms
2000	1 sec	8	$\leq$ 281 ms	30 ms
2000	1 sec	16	$\leq$ 375 ms	72 ms
2000	1 sec	32	$\leq$ 750 ms	83 ms
2000	1 sec	64	$\leq$ 1610 ms	100 ms
2000	1 sec	128	$\leq$ 4546 ms	182 ms
5000	4 sec	1	1 ms	1 ms
5000	4 sec	2	1 ms	1 ms
5000	4 sec	4	$\leq$ 16 ms	2 ms
5000	4 sec	8	$\leq$ 175 ms	16 ms
5000	4 sec	16	$\leq$ 184 ms	16 ms
5000	4 sec	32	$\leq$ 859 ms	67 ms
5000	4 sec	64	$\leq$ 1859 ms	68 ms
5000	4 sec	128	$\leq$ 6765 ms	233 ms
5000	2 sec	1	1 ms	1 ms
5000	2 sec	2	1 ms	1 ms
5000	2 sec	4	$\leq$ 16 ms	2 ms
5000	2 sec	8	$\leq$ 173 ms	16 ms
5000	2 sec	16	$\leq$ 185 ms	16 ms
5000	2 sec	32	$\leq$ 922 ms	105 ms
5000	2 sec	64	$\leq$ 2172 ms	439 ms
5000	2 sec	128	$\leq$ 8610 ms	2151 ms
5000	1 sec	1	1 ms	1 ms
5000	1 sec	2	1 ms	1 ms
5000	1 sec	4	$\leq$ 360 ms	32 ms
5000	1 sec	8	$\leq$ 485 ms	35 ms
5000	1 sec	16	$\leq$ 1219 ms	436 ms
5000	1 sec	32	$\leq$ 1640 ms	534 ms
5000	1 sec	64	$\leq$ 5203 ms	712 ms
5000	1 sec	128	$\leq$ 17828 ms	5861 ms

I risultati mostrano che il sistema, dato un certo dataset e tempo di campionamento, aumenta i tempi medi di risposta all'aumentare del numero di query consumatore e sopratutto aumenta drasticamente la varianza con cui tali risposte

vengono fornite. Questo comportamento viene causato dal fatto che i thread, per poter leggere, devono acquisire un accesso esclusivo allo store condiviso, bloccando di conseguenza la lettura agli altri thread. Inoltre, se si diminuisce il tempo di campionamento, il produttore esegue dei lock sullo store più spesso, facendo aumentare i tempi di attesa dei consumatori e di conseguenza aumentando i tempi di risposta e la varianza. Inoltre il produttore, dovendo leggere da file, peggiora drasticamente i tempi del sistema in quanto i tempi di accesso al disco sono di qualche ordine di grandezza superiori ai tempi di accesso alla RAM. Questo problema potrebbe essere risolto creando differenti store 'gemelli', in mo-

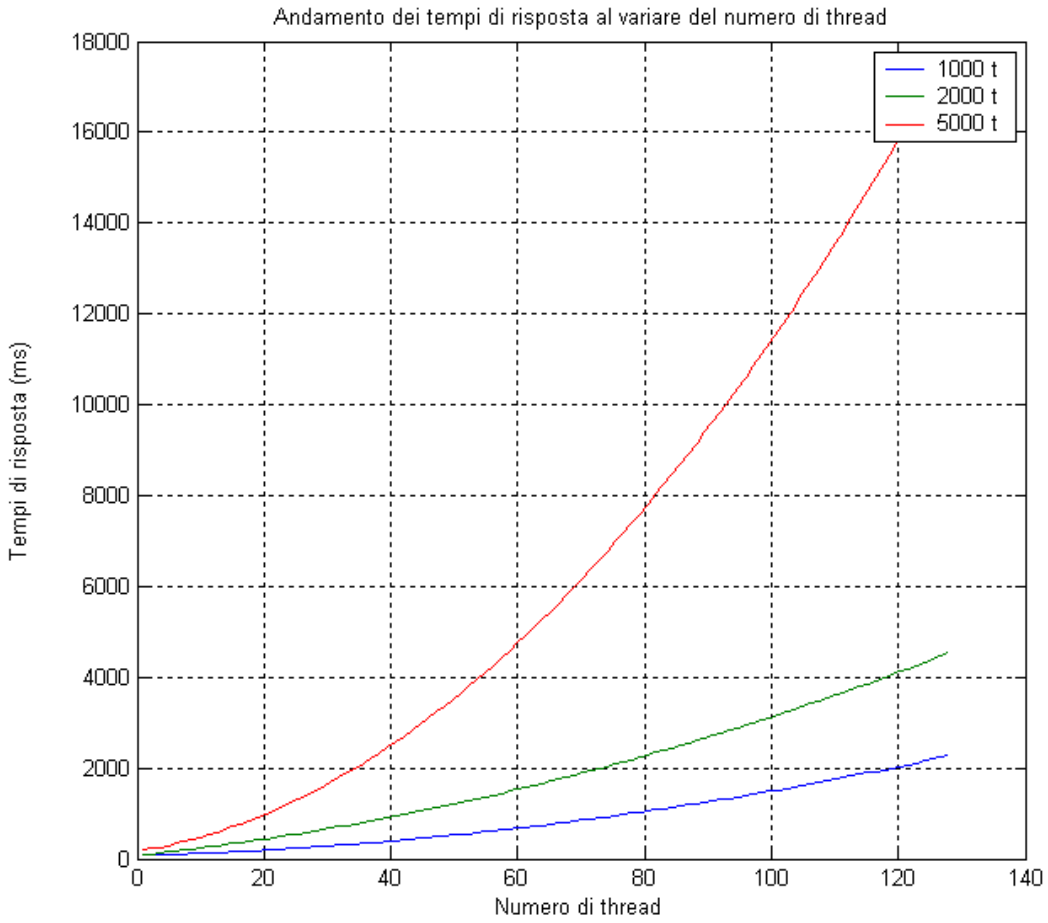


Figura 5.1: Andamento dei tempi di risposta massimi al variare del numero di thread e del dataset. Si vede che, aumentando la quantità di informazioni che il produttore legge da file e il numero di thread, i tempi di risposta aumentano. Il polling time del produttore è uguale a 1 secondo.

do da permettere ai thread di leggere piu spesso i dati senza dover aspettare, ma bisognerà gestire la coerenza dei dati tra gli store gemelli e sopratutto bisognerà gestire il tradeoff tra consumo di memoria e tempi di risposta (più store gemelli tengo in memoria più diminuisco i tempi di risposta ma aumento il consumo di memoria). La soluzione migliore dipenderà quindi dal contesto applicativo nel quale ci si dovrà inserire e dalle caratteristiche hardware della macchina che dovrà ospitare l'applicazione.

5.2.3 Performance Testing

In questa tipologia di test si è cercato di valutare l'andamento asintotico dell'algoritmo che legge i dati i dati dallo store e calcolarne le prestazioni al variare della quantità di dati in memoria.

Data n la quantità di tuple presenti in memoria e k la dimensione della finestra temporale della query, l'algoritmo esegue una scansione della lista di puntatori a blocchi tramite un doppio ciclo `for`, in cui il primo ciclo scandisce i directors mentre il secondo scandisce il blocco puntato dal director. In totale i due cicli iterano su d directors e b blocchi, in cui $d \cdot b \approx n$. In prima battuta, considerando un algoritmo banale che non tiene conto delle caratteristiche della struttura dati sottostante, verrebbe da dire che le prestazioni di tale algoritmo sono sempre $O(n)$ (lineare nel numero di elementi presenti nello store) per qualsiasi query in quanto è necessario scandire tutti gli elementi della lista per trovare le tuple che soddisfano le condizioni imposte dalla query.

Considerando però le caratteristiche della struttura dati si riescono a migliorare le prestazioni di tale algoritmo per il caso medio, evitando i confronti inutili. Nel capitolo precedente abbiamo spiegato che gli elementi nella struttura dati vengono sempre mantenuti in ordine temporale, per cui procedendo dall'inizio alla fine della lista si trovano elementi sempre più recenti. Sfruttando questa caratteristica si riescono a diminuire il numero confronti da effettuare, ovvero ci basterà scorrere la lista al contrario e fermarsi non appena si incontra un elemento che non entra nella finestra temporale del consumatore, in quanto tutti gli elementi a lui a precedenti saranno ancora più vecchi e quindi non necessari. In questo modo, le prestazioni dell'algoritmo saranno $O(k)$ (lineare nella dimensione della finestra), che nel caso peggiore (finestra temporale che include tutti gli n elementi nello store) diventa $O(n)$, ma che in media è $< O(n)$.

Sono stati eseguiti vari test per valutare gli effettivi guadagni dell'algoritmo rispetto alla soluzione banale e considerando il limite massimo che il sistema riesce a gestire di 180.000 tuple e sottoponendo al sistema varie query con differenti finestre temporali si è ottenuto che nel *worst case* (caso peggiore, $n = k$) l'algoritmo fornisce le risposte in 72 ms mentre nel *best case* in 16 ms, facendo risparmiare quindi 56 ms rispetto alla soluzione banale. In media invece le risposte vengono fornite in 32 ms. Il guadagno non è molto rispetto l'algoritmo banale, ma se dovessimo considerare una sistema che riesca a gestire un numero di tuple molto più elevato la differenza potrebbe diventare notevolmente più marcata.

La soluzione lineare è l'unica soluzione possibile se si vuole avere la massima accuratezza nelle risposte in quanto è necessario scandire tutti i dati per calcolare i risultati. Se l'applicazione finale accetta però un certo margine di errore, allora in uno sviluppo futuro potranno essere implementati algoritmi pseudo-casuali o di approssimazione che migliorano i tempi di risposta ma introducono un certo errore.

5.3 Conclusioni e sviluppi futuri

Durante questa Tesi di Laurea è stato affrontato lo studio delle problematiche relative alla gestione real-time di flussi di dati continui, ovvero dei Data Stream Management System. Tali sistemi sono richiesti in tante applicazioni del mondo reale, come la gestione dei traffico urbano, l'analisi finanziaria, il network management, le reti di sensori, ecc. Nonostante il vasto campo di possibili applicazioni, la Tesi si inserisce in un progetto ben preciso, denominato PEGASUS, il cui obbiettivo finale è la gestione intelligente del traffico urbano. Il controllo dei veicoli viene effettuato installando a bordo di ciascun veicolo un dispositivo denominato OBU, in grado di comunicare alla centrale operativa varie informazioni sul veicolo stesso come la velocità, le accellerazioni, la posizione, ecc. La centrale di controllo dovrà quindi gestire in real time un flusso di dati molto elevato proveniente da vari veicoli sparsi per la città, in modo da offrire servizi per la smart navigation e l'assistenza in caso di incidenti.

Per poter quindi gestire in maniera efficiente dei dati in streaming su cui vengono richiesti servizi real time si devono affrontare numerose problematiche e si sono dovuti studiare degli innovativi sistemi per la gestione degli stream di dati

chiamati Data Stream Management System. Questi sistemi sono profondamente diversi rispetto ai tradizionali DBMS, in cui i dati sono essenzialmente 'statici' e le query sono generalmente one shot, per cui è stato necessario, in prima battuta, uno studio approfondito delle caratteristiche tipiche di un DSMS e dei requisiti di performance richiesti dall'applicazione finale. Le applicazioni che sfruttano i DSMS hanno tipicamente requisiti di performance molto stringenti e devono operare su uno stream di dati potenzialmente infinito, pertanto i dati non possono essere gestiti come in un DBMS tradizionale. In particolare, la prima distinzione che è stata fatta è quella tra dato 'recente' e dato 'storico'. I dati recenti sono tipicamente impiegati per servizi hard real time, in cui i vincoli di performance sono molto stringenti, mentre i dati storici vengono impiegati in un secondo momento per effettuare previsioni (ad esempio sul traffico futuro) o analisi statistiche. Secondariamente, per quel che riguarda la gestione dei dati recenti, si sono studiate ed analizzate differenti strutture dati e politiche che regolano il passaggio da una struttura ad un altra in base alle caratteristiche delle query presenti nel sistema, in modo da garantire la massima efficienza nel recupero dei dati.

Quasi tutti i DSMS odierni hanno però una forte limitazione, ovvero accoppiano la parte di gestione dei dati con quella di query processing. Questa soluzione non solo crea un sistema ad hoc e poco flessibile, ma diminuisce le possibilità di ottimizzazione in base alle caratteristiche dell'applicazione finale.

E' stato quindi progettato ed implementato un innovativo DSMS, in cui la parte di gestione dei dati è ben separata rispetto alla parte di query processing engine, in modo da garantire la massima flessibilità e ottime possibilità di ottimizzazione. Per fare questo, sono stati analizzati una serie di parametri che possono essere sfruttati per migliorare le prestazioni dello storage manager, come ad esempio le caratteristiche delle query presenti nel sistema o lo schema sui dati, ed è stata studiata l'interfaccia che permette al query engine di comunicare con lo storage manager in modo da poter recuperare i dati e comunicare nuove query consumatore. Il prototipo è stato sviluppato in Java sfruttando il DBMS PostgreSQL per la gestione dei dati storici.

Infine, sono stati eseguiti vari test per valutare le prestazioni e la scalabilità del sistema al variare del carico di lavoro e delle query presenti nel sistema.

Il lavoro prodotto si inserisce nell'ambito di una problematica ben più vasta di quanto è possibile trattare in una Tesi di Laurea e si presta di conseguenza a

molti e interessanti futuri sviluppi sotto diversi aspetti. In primo luogo, anzichè leggere i valori delle OBU da file ad intervalli regolari, bisognerà gestire un flusso di dati asincrono in arrivo dalla rete in cui potrebbero essere presenti pacchetti danneggiati o mancanti. Secondariamente, bisognerà sviluppare un motore di parsing in grado di analizzare delle query SQL-like in cui saranno presenti nuovi concetti relativi alla gestione di dati continui, come la finestra temporale o il tempo di riesecuzione, e dovranno essere implementati meccanismi di ottimizzazione delle query (query plan dinamici, adaptivity, ecc.) e degli algoritmi (soluzioni approssimate, load shedding, ecc.). Inoltre bisognerà implementare tutti i servizi necessari per la smart navigation e la gestione del traffico, come ad esempio il calcolo del percorso minimo, l'aggiornamento dinamico del percorso ottimo in base alle caratteristiche del traffico, il rilevamento di incidenti tramite tecniche di data stream mining, gli EOI, e tanti altri. Per concludere, bisognerà implementare un recommender system collaborativo per la gestione dei POI segnalati dall'utente, in modo da offrire dinamicamente a ciascun automobilista dei punti sulla mappa che con grande probabilità saranno di loro interesse.

Tutte le parti del lavoro che mi sono state proposte sono state portate a termine in maniera completa e concreta ed hanno soddisfatto ampiamente le esigenze dei Professori. Il progetto PEGASUS, nella sua completezza, è un progetto estremamente vasto che richiederà all'università e ai suoi collaboratori esterni qualche anno di lavoro tra progettazione, implementazione e testing. Tuttavia il mio contributo è stato di notevole importanza ed utilità e spero verrà preso in considerazione per gli sviluppi futuri. Questa esperienza di Tesi è stata decisamente positiva e mi ha permesso di capire le logiche di funzionamento e le complessità che si nascondono dietro il mondo dei Data Stream Management System.

Bibliografia

[1] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, C. Erwin, Eduardo F. Galvez, M. Hatoun, Anurag Maskey, Alex Rasin, A. Singer, Michael Stonebraker, Nesime Tatbul, Ying Xing, R. Yan, and Stanley B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139, 2003.

[2] G. Adomavicius and A. Tuzhilin. Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions. *IEEE TKDE*, 17(6):734–749, 2005.

[3] Mark Crovella Christophe Diot Eric D. Kolaczyk Anukool Lakhina, Konstantina Papagiannaki and Nina Taft. Structural Analysis of Network Traffic Flows. *Boston University*, 57(3):1–12, 2004.

[4] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. STREAM: The Stanford Data Stream Management System. In M. Garofalakis, J. Gehrke, and R. Rastogi, editors, *Data Stream Management: Processing High-Speed Data Streams*. Springer, 2007.

[5] Arvind Arasu, Mitch Cherniack, Eduardo F. Galvez, David Maier, Anurag Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear Road: A Stream Data Management Benchmark. In *Proc. of VLDB*, pages 480–491, 2004.

[6] Arvind Arasu and Jennifer Widom. A Denotational Semantics for Continuous Queries over Streams and Relations. *Stanford University*, 57(3):1–8, 2004.

[7] I. Botan, G. Alonso, P. M. Fischer, D. Kossmann, and N. Tatbul. Flexible and scalable storage management for data-intensive stream processing. In *Proc. of EDBT*, pages 934–945, 2009.

[8] R. Bruno, M. Conti, and E. Gregori. Mesh networks: commodity multihop ad hoc networks. *IEEE Communications Magazine*, 43(3):123–131, 2005.

[9] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc of SIGMOD*, pages 379–390, 2000.

[10] D.J. Cook and John Wiley S.K. Das. Wireless Sensor Network. *University of Texas*, 57(3):1–18, 2004.

[11] R. Ding and Q. Zeng. A Clustering-based Multi-channel Vehicle-to-Vehicle (V2V) Communication System. In *Proc. of ICUFN*, pages 83–88, 2009.

[12] S. Goel, T. Imielinski, and K. Ozbay. Ascertaining Viability of WiFi based Vehicle-to-Vehicle Network for Traffic Information Dissemination. In *Proc. of IEEE ITSC*, 2004.

[13] Lukasz Golab and M. Tamer Ozsu. Issues in Data Stream Management. *Natural Sciences and Engineering Research Council*, 57(3):5–14, 2003.

[14] C. Guestrin, P. Bodik, R. Thibaux, M. Paskin, and S. Madden. Distributed Regression: an Efficient Framework for Modeling Sensor Network Data. In *Proc. of IPSN*, pages 1–10, 2004.

[15] Brian Babcock Shivnath Babu Mayur Datar Rajeev Motwani Jennifer-Widom. Models and Issues in Data Stream Systems. *Stanford University*, 57(3):1–16, 2005.

[16] O. Kennedy, C. Koch, and A. J. Demers. Dynamic Approaches to In-network Aggregation. In *Proc. of ICDE*, 2009.

[17] U. Lee and M. Gerla. A survey of urban vehicular sensing platforms. *Computer Networks*, 54(4):527–544, 2010.

[18] A. Melki, S. Hammadi, Y. Sallez, T. Berger, and C. Tahon. Advanced approach for the public transportation regulation system based on cybercars. *RAIRO-Oper. Res.*, 44(1):85–105, 2010.

[19] S. E. Middleton, N. R. Shadbolt, and D. C. De Roure. Ontological User Profiling in Recommender Systems. *ACM TOIS*, 22(1):54–88, 2004.

[20] Arkady Zaslavsky Mohamed Medhat Gaber and Shonali Krishnaswamy. Mining Data Streams: A Review. *SIGMOD Record*, 57(3):18–26, 2005.

[21] C. Ré, J. Letchner, M. Balazinska, and D. Suciu. Event queries on correlated probabilistic streams. In *Proc. of SIGMOD*, pages 715–728, 2008.

[22] Amol Deshpande Michael J. Franklin Joseph M. Hellerstein Wei Hong Sailesh Krishnamurthy Sam Madden Vijayshankar Raman Fred Reiss Sirish Chandrasekaran, Owen Cooper and Mehul Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. *University of California, Berkeley*, 57(3):1–12, 2003.

[23] L. Speičvcys, C.S. Jensen, and A. Kligys. Computational data modeling for network-constrained moving objects. In *Proc. of GIS*, pages 118–125, 2003.

[24] Kyuseok Shim Sudipto Guhay, Nick Koudasz. Approximation and Streaming Algorithms for Histogram Construction Problems. *IEEE International Conference on Data Engineering,*, 57(3):1–42, 2002.

[25] Y. Yao and J. Gehrke. Query Processing in Sensor Networks. In *Proc. of CIDR*, 2003.

[26] Yong Yao and Johannes Gehrke. The Cougar Approach to In-Network Query Processing in Sensor Network. *Cornell University*, 57(3):9–18, 2002.

[27] P. Zhuang, Q. Qi, and Y. Shang. Wireless sensor networks in intelligent transportation systems. *Wireless Communications and Mobile Computing*, 9(3):287–302, 2009.