

La persistenza dei dati nelle piattaforme di cloud computing: esperimenti su Google App Engine.

Data persistence in cloud computing platforms: experiments on Google App Engine

Laureando: Pieruigi Conte

Relatore: prof. Massimo Maresca

*We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity,
and especially because it produces objects of beauty.
A programmer who subconsciously views himself as an artist
will enjoy what he does and will do it better.
[Knuth, Donald Ervin]*

1	Nozioni introduttive	3
1.1	Cloud computing	3
1.1.1	SaaS - Software as a Service	5
1.1.2	PaaS - Platform as a Service	7
1.1.3	IaaS - Infrastructure as a Service	9
1.2	Persistenza	11
1.2.1	DBMS	11
1.2.2	ORM	13
1.2.3	Hibernate	14
1.2.4	Eclipse_Link	14
1.2.5	JPA	14
1.3	Apache Tomcat	15
1.3.1	Java Servlet	15
1.3.2	JavaServer Pages	16
2	Google App Engine	19
2.1	Architettura di App Engine	19
2.2	Runtime environment	21
2.2.1	Frontends	21
2.2.2	Static file servers	22
2.2.3	App servers	22
2.2.4	App master	23
2.3	Datastore	23
2.3.1	Bigtable	25
2.3.2	Entities & keys	26
2.3.3	Query e indici	27
2.3.4	Transazioni e <i>entity groups</i>	27
2.3.5	API di accesso	29
2.4	Preparazione dell'ambiente di lavoro	29
2.4.1	Registrazione di un Google Account	29
2.4.2	Installazione di Eclipse IDE	30
2.4.3	Configurazione di Eclipse IDE	31
3	Test funzionali sul datastore App Engine	33
3.1	Persistenza di una singola entità	33
3.1.1	Descrizione	33
3.1.2	Creazione del progetto	33
3.1.3	Creazione dell'entità	35
3.1.4	Implementazione del provider CRUD generico	35
3.1.5	Uso di EclipseLink	36

3.1.6	Uso di Hibernate	37
3.1.7	Migrazione su App Engine	37
3.1.8	Osservazioni	40
3.2	Peristenza di una relazione uno-a-molti	41
3.2.1	Descrizione	41
3.2.2	Modifiche alle entità	41
3.2.3	Modifiche al codice di testing	41
3.2.4	Problema: la gestione delle chiavi di AppEngine	43
3.2.5	Osservazioni	46
3.3	Peristenza di una relazione molti-a-molti	47
3.3.1	Descrizione	47
3.3.2	Modifica alle entità e all'Entity Generator	47
3.3.3	Risultati	47
3.3.4	Osservazioni	48
4	Utilizzo di JDBC su App Engine	51
4.1	Emulazione di JDBC tramite JPQL	51
4.1.1	Descrizione	51
4.1.2	Nozioni preliminari	52
4.1.3	Le classi di <i>bridging</i>	53
4.1.4	Analisi delle query	54
4.1.5	Osservazioni	57
4.2	Architettura	59
4.2.1	Descrizione	59
4.2.2	JSON	59
4.2.3	nembo-gae	61
4.2.4	nembo-db	61
4.2.5	jdbc-on-nembo	62
5	Test prestazionali	63
5.1	Gestione delle sessioni	63
5.1.1	Descrizione	63
5.1.2	L'interfaccia HttpSession	63
5.1.3	Supporto su AppEngine	64
5.1.4	Osservazioni	64
5.2	Inserimenti su datastore	66
5.2.1	Descrizione	66
5.2.2	gae-time-datastore	67
5.2.3	local-time-datastore	68
5.2.4	I test effettuati	69
5.2.5	Osservazioni sulle latenze di inserimento	70
5.2.6	Osservazioni sui tempi di risposta della servlet	71
5.2.7	Osservazioni sulle istanze attivate	72
5.3	Inserimenti su <i>nembo</i>	73

5.3.1	gae-time-nembo	74
5.3.2	local-time-nembo	75
5.3.3	Osservazioni sulle latenze di inserimento	75
5.3.4	Osservazioni sui tempi di risposta della servlet	76
5.3.5	Osservazioni sulle istanze attivate	76
6	Conclusioni	79
A	Codice sorgente	83
A.1	test-crud-single	83
A.2	test-crud-one-to-many	90
A.3	test-crud-many-to-many	97
A.4	test-crud-jdbc	102
A.5	test-session	107
A.6	gae-time-datastore	110
A.7	local-time-datastore	113
B	Grafici	115
B.1	local-time-datastore ($L=100$)	115
B.2	local-time-datastore ($L=10$)	126
B.3	Problemi dovuti al proxy	134

Tra le proposte fortemente innovative che nell'ultimo periodo stanno rivoluzionando il mondo dei servizi informatici, il *cloud computing* rappresenta certamente una novità degna di nota per le sue molteplici peculiarità.

Ad un primo approccio, il cloud computing dovrebbe trasformare l'accesso alle risorse informatiche, avvicinandolo ad un utilizzo simile alla rete di distribuzione dell'elettricità: secondo questa idea, un generico utente potrebbe semplicemente accedere al *cloud*, ossia alla rete internet, in modo analogo al collegamento alla rete tramite presa elettrica.

Una volta collegato, egli dovrebbe poter accedere a dei generici servizi distribuiti (tra cui il calcolo, lo storage, ecc...), semplicemente pagando per l'effettivo servizio utilizzato, ignorando dove e come questo servizio viene effettivamente realizzato, e senza la necessità di predisporre e gestire una propria infrastruttura informatica.

Sotto questa definizione è possibile comprendere una serie molto ampia di architetture e di servizi distribuiti, tanto che si rende necessario fin da subito una caratterizzazione rigorosa delle possibili varianti: in particolare, la distinzione più utilizzata si basa sul *service model*. In base alla tipologia di servizio offerto, possiamo distinguere i tre approcci:

SOFTWARE AS A SERVICE , dove la risorsa messa a disposizione è un software ospitato su una piattaforma cloud;

PLATFORM AS A SERVICE , dove il provider mette a disposizione una piattaforma software atta ad ospitare applicazioni (solitamente applicazioni web), create secondo precise specifiche;

INFRASTRUCTURE AS A SERVICE , in cui l'utente ha il più alto grado di controllo, avendo a disposizione una infrastruttura hardware in cui ha facoltà gestire in modo più o meno vincolato il software.

La nostra attenzione sarà essenzialmente rivolta all'approccio *PaaS*, il quale unisce l'innovativo trend del *cloud* con il più consolidato trend dell'utilizzo delle piattaforme informatiche per lo sviluppo di applicazioni.

In questo settore, tra le proposte più interessanti troviamo App Engine di Google e Microsoft Azure Platform di Microsoft. In particolare, App Engine rappresenta un caso di studio molto interessante, in cui la costruzione di una piattaforma per applicazioni web ha portato a soluzioni che, in alcuni casi, si discostano di molto dalle soluzioni comunemente utilizzate dagli sviluppatori.

Proporremo quindi uno studio delle funzionalità e dell'organizzazione architetturale di App Engine, evidenziando come si sia voluta proporre una piattaforma distribuita ad elevata scalabilità, cercando in molti casi di

sfruttare tecnologie già esistenti all'interno della compagnia, anche al costo di imporre approcci alternativi e limitazioni nelle funzionalità.

Il caso più emblematico è certamente il *Datastore*, che gestisce la persistenza degli oggetti sul cloud: si tratta di un database distribuito, che gestisce *entities*, basato sulla tecnologia *BigTable* che rappresenta la piattaforma di persistenza storica di Google.

L'approccio tradizionale alla persistenza, anche all'interno dei web service, prevede invece l'utilizzo di DBMS (**D**ATA**B**ASE **M**ANAGEMENT **S**YSTEM) relazionali, e in particolare l'accesso tramite le API (**A**PPPLICATION **P**ROGRAMMING **I**NTERFACE) JDBC (**J**AVA **D**ATA**B**ASE **C**ONNECTIVITY) o tramite ORM (**O**BJECT-**R**ELATIONAL **M**APPING).

In questo senso lo sviluppatore si trova innanzitutto a dover affrontare un probabile cambio nel proprio paradigma di programmazione, nel caso in cui utilizzi abitualmente JDBC; nel caso in cui possa vantare la conoscenza di JPA (**J**AVA **P**ERSISTENCE **A**PI), dovrà comunque attenersi alle limitazioni imposte, relative alla struttura delle transazioni, alle associazioni tra dati, alla gestione delle chiavi.

Il nostro lavoro andrà inizialmente ad evidenziare come l'approccio del programmatore alla persistenza debba confrontarsi con scelte progettuali e limitazioni imposte, anche attraverso la stesura di semplici applicazioni dimostrative: si tratta quindi di un test che evidenzia le *funzionalità* di cui può disporre uno sviluppatore che utilizzi la persistenza.

Successivamente, si è cercato di verificare come del codice già esistente, basato sulle API JDBC, possa essere riutilizzata su App Engine senza modifiche radicali al codice. Dato per assunto che tali API non sono supportate in modo nativo, si è cercato di proporre delle metodologie per inserire uno strato software intermedio, in grado di emularne i metodi di accesso.

Il primo approccio, relativamente elementare, prevedeva lo sfruttamento del linguaggio JPQL (**J**AVA **P**ERSISTENCE **Q**UERY **L**ANGUAGE), supportato da App Engine (sia pur con qualche limitazione), che per la sua somiglianza con SQL (**S**TRUCTURED **Q**UERY **L**ANGUAGE) poteva prestarsi ad una traduzione delle query eseguite dal programmatore.

Il secondo approccio, più elaborato ma con una maggiore potenza, ha portato a sviluppare una libreria di emulazione di JDBC, chiamata *nembo*, che trasferisse le chiamate ad un DBMS relazionale, ospitato all'esterno di App Engine, i cui metodi di accesso sono stati modellati sotto forma di web service.

Una volta verificato che l'introduzione di *nembo* consentiva effettivamente di supportare le funzionalità garantite da JDBC, si sono realizzati una serie di test prestazionali, relative alle latenze di scrittura e al comportamento della servlet in toto, relativamente a latenza e scalabilità. Tali test hanno permesso di confrontare i risultati prestazionali ottenuti, paragonandoli ad analoghi test in cui la persistenza veniva realizzata sul datastore App Engine.

1 NOZIONI INTRODUTTIVE

Iniziamo questa trattazione formalizzando alcuni termini che ricorreranno frequentemente nei capitoli successivi: proponiamo quindi una categorizzazione rigorosa delle tipologie di *cloud computing*, con esempi relativi allo stato dell'arte tra le proposte sul mercato, oltre a illustrare una serie di strumenti software che verranno utilizzati nei test successivi.

1.1 Cloud computing

Ovviamente il primo termine al quale vogliamo dare un significato è *cloud computing*, ricorriamo quindi alla definizione che ne dà il NIST¹ nella sua *Definition of cloud computing* [1].

Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model promotes availability and is composed of five essential *characteristics*, three *service models*, and four *deployment models*.

Il cloud computing definisce quindi un modello di accesso via rete ad un insieme di risorse computazionali condivise e configurabili: da questa descrizione, emerge che il termine “cloud” è utilizzato come una metafora per Internet, e rappresenta la volontà di astrarre dalle strutture e dalle infrastrutture sottostanti.

Ne consegue che il modello di dislocazione spaziale delle risorse non appare più significativo, ed esso deve essere inteso come distribuito “on the cloud”, ad accesso il più possibile on-demand, con una buona analogia a quanto avviene oggi con la rete distributiva di energia elettrica, dove è possibile accedere alle risorse elettriche semplicemente collegando una spina alla corretta presa, ignorando dove e come l'energia viene prodotta.

Seguendo la definizione, le 5 *caratteristiche* che caratterizzano un modello di *cloud computing* sono:

On-demand self-service: la richiesta di risorse avviene in modalità on-demand, ossia la risorsa dovrebbe poter essere assegnata all'utente che ne fa richiesta in modo automatico, senza necessità di interazione umana con ciascun provider.

¹Il NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY è un'agenzia dell'U.S. Department of Commerce, con la *mission* di promuovere l'innovazione e la competitività industriale attraverso il progresso della scienza delle misurazioni, degli standard e della tecnologia [2]

Broad network access: il canale di accesso alle risorse è rappresentato dalla rete, attraverso meccanismi standardizzati che consentano l'utilizzo di client eterogenei.

Resource pooling: le risorse del provider sono organizzate in pool e disponibili secondo il modello multi-tenant² dove le risorse sia fisiche sia virtuali sono dinamicamente riassegnate sulla base della richiesta del client. Il tutto è indipendente dalla dislocazione spaziale, nel senso che l'utente generalmente non ha ne' controllo ne' conoscenza sull'effettiva locazione delle risorse, anche se può avere un certo controllo ad un livello più alto (regione geografica o datacenter).

Rapid elasticity: le risorse devono essere fornite in modo rapido ed elastico all'aumentare del carico e rilasciate rapidamente al suo diminuire: l'utente ha l'illusione di una capacità infinita delle risorse e di una disponibilità per una qualsiasi quantità in qualsiasi momento.

Measured Service: il controllo e l'ottimizzazione delle risorse fa riferimento a una qualche metrica, correlata col tipo di servizio offerto. L'utilizzo di risorse può essere monitorato, controllato e visualizzato in modo trasparente sia dal provider che dall'utente.

E' importante osservare come il cloud computing si ponga come un paradigma che riguarda la produzione, il consumo e la distribuzione di generici servizi, e sia fortemente connesso con la necessità di sottrarre all'utente finale la preoccupazione sull'effettiva realizzazione dei meccanismi di virtualizzazione e scalabilità.

La descrizione del paradigma è volutamente molto generica - *cosa si configura come risorsa? quali sono i punti di accesso al cloud? come definisco un servizio, e come definisco fornitore e fruitore di tale servizio?* - di conseguenza ci si appoggia ai tre *service models* che definiscono meglio tali dettagli, che verranno presentati nelle prossime sottosezioni.

Viceversa, la descrizione dei *deployment models* riguarda l'organizzazione logistica della struttura di cloud, e prevede i seguenti scenari:

Private cloud: L'infrastruttura viene costruita ad uso esclusivo di una organizzazione: può essere mantenuta dall'organizzazione stessa o da una terza parte, e può essere ospitata sia all'interno che fuori sede.

Community cloud: L'infrastruttura viene condivisa da diverse organizzazioni, ma supporta una specifica comunità identificata da un qualche concetto comune (mission, requisiti di sicurezza, policy,...). Le considerazioni su mantenimento e dislocazione sono le stesse del caso precedente.

²La *multitenancy* è un modello architetturale dove la singola istanza del software serve più *tenants* (inquilini), richiede un partizionamento logico dei dati e delle configurazioni in modo da adattare la singola istanza alle esigenze e al contesto richiesto dal singolo utente.

Public cloud: L'infrastruttura è disponibile per uso generale o per una larga utenza industriale, sotto la proprietà di una organizzazione che distribuisce servizi di cloud.

Hybrid cloud: L'infrastruttura è formata dalla composizione di più strutture appartenenti agli scenari precedenti, che rimangono comunque entità separate ma utilizzano tecnologie standardizzate o proprietarie al fine di garantire portabilità di dati e applicazioni.

La catalogazione dei *service models* realizza una classificazione basata sulla tipologia del servizio offerto: ovviamente tali classi non sono nettamente separate e non è infrequente imbattersi in casi che si posizionano tra due classi adiacenti.

Nelle tre sottosezioni che seguono verranno illustrati gli spetti caratterizzanti delle tre classi, definite come “**SOFTWARE AS A SERVICE**”, “**PLATFORM AS A SERVICE**” e “**INFRASTRUCTURE AS A SERVICE**”.

1.1.1 SaaS - Software as a Service

Nel modello SaaS il servizio, ossia l'oggetto del rapporto tra fornitore e fruitore, è il software: il provider rende disponibile l'uso di una applicazione ospitata su una infrastruttura di cloud. In questo caso, esiste una licenza che regola l'accesso alla web-application, che si configura come un servizio on-demand.

L'utente non è in grado di gestire o controllare l'infrastruttura di cloud, che include rete, server, sistemi operativi, spazio su disco, fino ad arrivare alle specifiche dell'applicazione, con l'eccezione di configurazioni limitate e specifiche dell'utente.

In questa categoria rientra il CRM³ Salesforce.com (www.salesforce.com).

Il CRM di Salesforce.com

Salesforce.com è stata fondata nel 1999 come una compagnia specializzata nel SaaS, e si è successivamente espansa fino ad essere quotata in borsa nel 2004. Ad oggi i loro prodotti servono una base di circa 92.000 clienti [3].

Vediamo in breve in cosa consiste il prodotto CRM di SalesForce.com, attraverso un riassunto di quanto affermato sul loro sito [4].

Che cos'è un CRM?

Ogni giorno, gli operatori commerciali intrattengono rapporti con i clienti e presentazioni relative all'azienda e ai prodotti offerti. Il Customer Relationship Management (CRM) rappresenta l'insieme dei processi di business e delle applicazioni

³Il **CUSTOMER RELATIONSHIP MANAGEMENT** è un software di marketing che gestisce le relazioni tra azienda e cliente, focalizzandosi sia sull'acquisizione di nuovi clienti, sia sulla fidelizzazione o fornitura di servizi al parco clienti esistente.

correlate, che hanno lo scopo di coadiuvare la gestione di informazioni, attività e rapporti relativi ai clienti. Grazie al CRM, gli addetti alle vendite, al marketing e alla fornitura di servizi possono comprendere meglio il cliente in modo da fornire il corretto messaggio o risposta. Analizzando queste informazioni, è possibile prendere decisioni migliori, al fine di aumentare la redditività, diminuire il costo del servizio offerto, e mantenere alto il grado di soddisfazione del cliente.

CRM Cloud Applications

Al giorno d'oggi, il CRM si sta spostando nella piattaforma del cloud computing. Invece di comprare e mantenere costosi server e software per gestire rapporti e informazioni relative ai clienti, le compagnie possono utilizzare applicazioni basate sul web per supportare i loro CRM, ricevendo alti profitti su questi investimenti. Salesforce.com è una compagnia che può vantare una certa diversificazione nel settore del cloud computing, offrendo una applicazione CRM su cloud, oltre una piattaforma e una infrastruttura di cloud.

Per le vendite e il marketing

Per i manager del settore vendite, le applicazioni del CRM mettono a disposizione una visibilità in tempo reale sulle attività dei gruppi di lavoro, in modo da generare previsioni affidabili sulle vendite. Per i reparti di vendita, risulta più semplice gestire le informazioni, in modo da dedicare più tempo a rapporti produttivi con i clienti e meno alla gestione dei dati.

Per i venditori, è importante poter tracciare le vendite correlate a campagne di marketing sul sito web aziendale, tramite mail o sfruttando Google AdWords⁴. Il CRM consente di tracciare i percorsi web che hanno portato al contatto col cliente, per identificare i percorsi potenzialmente più promettenti, e per analizzare i punti di forza e le componenti migliorabili. [...]

Possiamo osservare che la tipologia del software si presta fortemente ad un approccio di tipo SaaS: il termine CRM riguarda infatti le applicazioni che regolano i rapporti di business tra l'azienda e il cliente. Rientrano in questa categoria attività relazionali quali il carico degli ordini, le statistiche sui prodotti, le interrogazioni sullo stato della produzione e delle consegne.

Dato che i clienti sono geograficamente sparsi, risulta naturale un approccio basato sulle web-applications, che in questo caso vengono supportate dal cloud: un'azienda che vuole dotarsi di un software di tipo CRM

⁴**GOOGLE ADWORDS** consente la pubblicazione di messaggi informativi o pubblicitari a pagamento, sfruttando la registrazione di determinate parole chiave che attivano la comparsa del messaggio all'interno dei risultati delle ricerche web.

può sfruttare il software sviluppato da Salesforce.com, semplicemente pagando una licenza che varia in base al numero degli utenti abilitati e ai servizi richiesti, con la possibilità di personalizzazioni ad hoc.

E' importante osservare che non vi è la necessità di installare software, in quanto il tutto è modellato come una applicazione web e risulta quindi accessibile tramite un semplice browser.

In conclusione non stupisce che oltre al prodotto SaaS, che rappresenta comunque il core business, vengono offerti anche servizi relativi alle piattaforme e alle infrastrutture - si rimanda alle sezioni immediatamente successive per maggiori dettagli - che risultano come una naturale evoluzione dell'offerta quando si dispone di una infrastruttura di cloud ormai matura e si vuole capitalizzarne l'accesso a diversi livelli.

1.1.2 PaaS - Platform as a Service

Nel modello PaaS il servizio fornito è la piattaforma informatica: il fornitore realizza una infrastruttura hardware, sulla quale organizza uno strato di software che consente al fruitore lo sviluppo di applicazioni (tipicamente web). L'utente può quindi distribuire sulla piattaforma le proprie applicazioni, create secondo linguaggi e strumenti supportati dal provider.

Si noti come l'approccio di mettere a disposizione una infrastruttura risultava ben consolidato già prima dell'introduzione del *cloud computing*: i vari application server, tra cui possiamo annoverare anche Tomcat, e le piattaforme (ad esempio .NET di Microsoft), consentono di semplificare la creazione di applicazioni business in quanto forniscono una interfaccia comune e indipendente dalla infrastruttura informatica.

In questo caso è il fruitore del servizio che costruisce gli ultimi strati software, ma fa riferimento al provider che gli mette a disposizione una piattaforma completa, svincolandolo dalla necessità di acquisto e manutenzione dell'hardware e della piattaforma software: si noti come è ragionevole che il fruitore non abbia nemmeno il controllo sul software del cloud, potendo agire solo su configurazione dell'ambiente di hosting.

In questa categoria rientrano Google App Engine (appengine.google.com) e Windows Azure Platform (www.microsoft.com/windowsazure), realizzato da Microsoft.

App Engine verrà trattato in modo esaustivo durante i capitoli successivi, sia sotto l'approccio architetturale che sotto l'approccio funzionale, per cui si rimanda ai capitoli successivi per dettagli.

Windows Azure Platform

Microsoft Windows Azure Platform rappresenta il prodotto che Microsoft offre nell'ambito del cloud computing: possiamo considerarlo un valido

concorrente ad App Engine, sia per la vastità dei servizi che si propone di offrire, sia per la solidità della compagnia che lo propone.

L'approccio che lo contraddistingue si differenzia in modo netto da App Engine: nel prodotto Microsoft possiamo osservare la ricerca di una presentazione *commerciale* nella proposta di un servizio cloud, volta a mettere a disposizione dello sviluppatore una suite che comprenda comunque le funzionalità che egli è abituato a utilizzare, a differenza della proposta Google che si presenta certamente più *innovativa*, a scapito magari dell'usabilità immediata.

Presenteremo ora una breve carrellata delle caratteristiche distintive, basandoci sulla presentazione web in [9], cercando quando possibile di evidenziarne le differenze rispetto ad App Engine, che consideriamo come riferimento.

Dal punto di vista della fornitura di servizi, possiamo notare come sia possibile caratterizzare le applicazioni ospitabili in tre categoria chiamate *roles*:

il Web Role prevede applicazioni web che sono ospitate in un web server IIS: si tratta quindi di un approccio puramente *PaaS*, molto simile a quello proposto da App Engine;

il Worker role prevede processi in background, che solitamente operano come supporto ad applicazioni nel Web role;

il Virtual Machine role si pone in antitesi alle due proposte precedenti, spostandosi verso una logica *IaaS*, in cui è possibile definire delle macchine virtuali in modo simile agli Amazon Web Services (presentati in sezione 1.1.3).

E' prevista una persistenza su database, tramite Microsoft SQL Azure, un DBMS scalabile basato su SQL Server, il principale DBMS relazionale di casa Microsoft. In questo modo è stato possibile mettere a disposizione degli utenti la persistenza su modello relazionale, unita alla possibilità di interagire con i dati tramite il linguaggio SQL, che sicuramente semplifica l'approccio per lo sviluppatore, e si contrappone in modo forte alla mancanza di un simile strumento per App Engine.

In modo simile ad App Engine e agli AWS, sono inoltre presenti una serie di servizi accessori su cloud, quindi con caratteristiche di scalabilità e distribuzione, che spaziano dallo storage, al caching, al trasferimento di messaggi, alle reti virtuali. Si tratta comunque di funzioni abbastanza diffuse in ambito cloud, che nella maggioranza dei casi sono previste anche dalle altre offerte prese in esame, sia in ambito *Paas* che *IaaS*.

1.1.3 IaaS - Infrastructure as a Service

Nel modello IaaS (a volte definito anche come HaaS - Hardware as a Service) il provider mette a disposizione l'infrastruttura computazionale, solitamente sotto forma di un ambiente virtualizzato. In questo caso il fruitore evita di sostenere l'acquisto di server, spazio di memorizzazione e dispositivi di rete, in quanto li ottiene dal fornitore come un servizio outsourced.

L'utente ha quindi a disposizione l'infrastruttura hardware, all'interno della quale ha facoltà di utilizzare software in modo arbitrario, a partire dal sistema operativo fino alle applicazioni: ha quindi il controllo completo delle componenti software e dello spazio su disco, mentre solitamente ha solo un controllo parziale delle componenti di rete.

In questa categoria rientrano gli Amazon Web Services.

AWS - Amazon Web Services

Gli AWS sono una suite di strumenti, realizzati sotto forma di *web services*, che realizzano una infrastruttura di cloud computing.



Figura 1.1: Logo degli Amazon Web Services

E' possibile usufruire dei vari servizi in maniera indipendente, secondo una filosofia *pay-as-you-go*, ma sono previste forme di interazione tra i vari prodotti (ad esempio tra le funzionalità di computing e quelle di storage), che consentono ai vari componenti di comunicare in modo organico al fine di massimizzarne le potenzialità.

Proponiamo ora una breve panoramica dei servizi offerti, tratta da [5], per maggiori dettagli si rimanda a [6]:

Amazon CloudFront Si tratta essenzialmente di un front-end che si occupa dell'accessibilità ai contenuti, ad elevate prestazioni e con distribuzione globale (si noti come questa definizione è volutamente molto generica, in quanto si prevede che tale applicazione sfrutti appieno i servizi complementari, e in tale accezione il content può assumere varie forme, ad esempio un file memorizzato nello storage oppure un risultato di una computazione).

Amazon Elastic Compute Cloud (Amazon EC2) Questo servizio mette a disposizione un ambiente virtualizzato, accessibile tramite un'interfaccia web, con funzionalità rivolte al *computing*.

Il sistema richiede l'utilizzo di una immagine virtualizzata (denominata *AMI – Amazon Machine Image*), che può essere creata ex novo in base alle applicazioni richieste oppure sfruttare template preconfigurati. Tale macchina può essere poi replicata su un certo numero di *istanze*, che rappresentano una sorta di unità di misura della potenza computazionale richiesta; è possibile configurare i criteri di sicurezza e accesso alle istanze, così come il deployment geografico per ridurre le latenze.

Ad oggi, la taglia standard di una istanza prevede 1,7 Gb di memoria, 1 *EC2 Compute Unit* (o ECU, equivalente ad un processore AMD Opteron del 2007 a 1.0-1.2 Ghz) e 160 Gb di storage locale su architettura a 32 bit.

Sono previste un certo numero di configurazioni a prestazioni via via crescenti, con la possibilità di privilegiare la potenza su parametri particolari (ad esempio richiedendo una elevata capacità di memoria o una elevata potenza di calcolo), ovviamente al crescere delle risorse impiegate cresce anche il costo orario da sostenere.

Altri parametri che influiscono sul costo orario dell'istanza sono il sistema operativo, la localizzazione geografica, l'impiego di banda da e verso il cloud, e l'utilizzo di servizi complementari del cloud come il server di Load Balancing.

Come ultimo dato interessante, possiamo notare che esistono tre tipologie di istanze, che vanno ad influire sul costo finale:

On-Demand Instances che sono per l'appunto allocate nel momento in cui il cloud rileva la richiesta di capacità computazionale e supportano la scalabilità dell'applicazione.

Reserved Instances assegnate permanentemente, che evitano il pagamento a consumo attraverso un canone fisso e supportano il carico medio dell'applicazione: ovviamente hanno un costo orario minore rispetto alla tipologia precedente.

Spot Instances rappresentano un compromesso tra le due categorie precedenti, consentono di gestire picchi di richiesta per cui è possibile accettare una risposta ritardata, e vengono allocate nei momenti in cui il cloud è poco utilizzato.

Amazon Relational Database Service (*Amazon RDS*) Espone un servizio economicamente conveniente e ridimensionabile per la gestione di dati su un database relazionale *MySQL* all'interno del cloud, gestendo automaticamente problematiche come il backup, la scalabilità e l'aggiornamento software.

Amazon SimpleDB Consente di eseguire interrogazioni su *dati strutturati* in tempo reale. Questo servizio offre le funzionalità principali di

un database, come le ricerche in tempo reale e la facilità di interrogazione su dati strutturati, senza comportare la stessa complessità operativa.

Amazon Simple Notification Service (*Amazon SNS*) Consente di propagare *notifiche* a partire dal cloud, verso altre applicazioni o utenti, in modo estremamente scalabile e flessibile, tramite un meccanismo di “push” che offre molteplici opzioni per quanto riguarda il protocollo utilizzato (email, HTTP, ...).

Amazon Simple Queue Service (*Amazon SQS*) E' un sistema di *queuing* ad elevata sicurezza, per distribuire il carico di lavoro in modo affidabile tra i vari processi dell'applicazione.

Amazon Simple Storage Service (*Amazon S3*) Consente di memorizzare grandi quantità di dati, mettendo a disposizione degli sviluppatori una piattaforma di *storage* scalabile, veloce, affidabile ed economica.

Amazon Virtual Private Cloud (*Amazon VPC*) Offre un semplice e sicuro *bridge* tra l'infrastruttura informatica dell'azienda e il cloud di AWS. In questo modo è possibile condividere risorse in modo sicuro attraverso una connessione Virtual Private Network (VPN), e includere i servizi del cloud nella cosiddetta rete interna dell'azienda, protetta da firewall e sistemi di prevenzione delle intrusioni.

1.2 Persistenza

Indipendentemente dall'utilizzo di una piattaforma di cloud o meno, l'esecuzione di un software incontra il concetto di persistenza quando si rende necessario *rendere disponibili questi dati anche al di fuori del tempo di vita dell'applicazione che li ha creati*.

La gestione della persistenza prevede quindi il salvataggio di tali dati su supporti non volatili: è possibile quindi memorizzarli in un semplice file, piuttosto che sfruttare basi di dati più evolute che semplifichino funzioni quali la ricerca o la gestione della concorrenza.

1.2.1 DBMS

Un **DATA**BASE **MANAGEMENT** **SYSTEM** è il sistema software che consente la creazione, la manutenzione e il funzionamento di una collezione di dati strutturati.

Secondo un approccio molto semplicistico, il DBMS gestisce, oltre alla ovvia memorizzazione dei dati in modo persistente, una serie di funzioni complementari come:

autorizzazioni : è possibile definire diversi account utenti, prevedendo per ciascuno di essi dei diritti o delle limitazioni, sia per l'accesso a un sottoinsieme del database, sia per la possibilità di eseguire determinate operazioni sui dati;

interrogazioni : vengono messi a disposizione degli strumenti per semplificare l'interrogazione dei dati, ossia l'estrazione di informazioni dal database;

integrità : l'integrità dei dati viene garantita attraverso una serie di vincoli, che comprendono ad esempio i vincoli di unicità delle chiavi primarie, i vincoli di integrità referenziale relativamente alle chiavi esterne, e soprattutto la regolamentazione dell'accesso concorrente ai dati al fine di evitare modifiche non coerenti. Per quanto riguarda la gestione della concorrenza, è possibile gestirla attraverso un banale *blocco record*, in cui un utente che tenta l'accesso a un record già utilizzato da un altro utente sperimenta un blocco nell'accesso che ha termine solo quando la prima modifica è completa, oppure attraverso le *transazioni*.

Per definizione, una transazione è una sequenza di operazioni che può risolversi con un successo o un insuccesso: nel caso di successo, *tutti* i risultati delle operazioni che la compongono risultano applicati, viceversa si ha un insuccesso e *nessuno* dei risultati viene applicato. Le transazioni devono rispettare quattro proprietà fondamentali, che devono essere implementate dal DBMS, solitamente raccolte nell'acronimo *ACID*:

atomicità : l'esecuzione della transazione è logicamente indivisibile, ossia non sono ammesse esecuzioni parziali del suo insieme di operazioni;

coerenza : sia all'inizio, sia al termine della transazioni il database deve trovarsi in uno stato coerente, ossia non devono essere violati eventuali vincoli di integrità e non devono verificarsi inconsistenze tra i dati memorizzati;

isolamento : la transazione non deve essere influenzata dall'esecuzione concorrente di altre transazioni;

durabilità : nel momento in cui si riconosce il successo di una transazione, i suoi effetti non possono essere persi, occorre quindi gestire accuratamente l'effettiva scrittura su memoria di massa dei dati.

Al momento, la maggioranza degli sviluppatori su basi di dati ha esperienza diretta sui DBMS relazionali (*RDBMS*), in cui i dati sono organizzati in tabelle (relazioni), suddivise in righe (tuple) logicamente individuate da

una chiave primaria, e le associazioni tra dati sono realizzate tramite la memorizzazione di una chiave esterna che rispecchia la chiave primaria del dato collegato.

La gestione dei dati nel modello relazionale può essere realizzata tramite lo standard SQL (**STRUCTURED QUERY LANGUAGE**)

L'architettura relazionale non è però una scelta univoca, sono presenti infatti molteplici varianti, di concezione più o meno innovativa, come gli ODBMS (**OBJECT DBMS**), che introducono una serie di variazioni al modello relazionale per consentire una maggiore aderenza della persistenza alla logica di programmazione ad oggetti.

1.2.2 ORM

Allo stato dell'arte, l'architettura DBMS più diffusa tra gli sviluppatori è il modello relazionale; allo stesso tempo, una porzione considerevole del software che sfrutta la persistenza segue una logica ad oggetti. La soluzione più immediata sarebbe utilizzare un DBMS ad oggetti, che potrebbe rispecchiare fedelmente in persistenza la logica degli oggetti generati dall'applicativo: tuttavia non sempre questa migrazione è possibile, o accettata dallo sviluppatore.

Per questo sono nati una serie di middleware di **OBJECT/RELATIONAL MAPPING**, che consentono di superare queste differenze tra i due paradigmi (a volte indicate come *Object-Relational Impedance Mismatch*, con una chiara ispirazione elettrotecnica), gestendo in modo automatico la conversione dei dati tra logica e persistenza.

Nel modello relazionale i dati sono organizzati in forma tabulare, mentre nella logica ad oggetti ci troviamo di fronte a un grafo interconnesso di oggetti. La conversione tra i due paradigmi comporta essenzialmente 5 problemi:

granularità : un oggetto può prevedere più classi rispetto al numero di tabelle corrispondenti nel database (in questo senso diciamo che il modello ad oggetti ha una granularità più fine), un esempio tipico è l'indirizzo;

ereditarietà : l'ereditarietà è un concetto basilare nel modello ad oggetti, che non trova una naturale traduzione nel modello relazionale, ad eccezione di alcune estensioni definite da particolari DBMS che soffrono però di una scarsa standardizzazione;

identità : nel modello relazionale, la definizione di identità tra due righe sfrutta la chiave primaria, mentre nel linguaggio ad oggetti è solitamente definita sia una nozione di *identity*, con comparazione per riferimento, sia una nozione di *equality* che in qualche modo coinvolge gli attributi dell'oggetto;

associazioni : la chiave primaria viene tradotta come un riferimento unidirezionale nel modello ad oggetti: tuttavia, se si vuole definire relazioni bidirezionali è necessario prevedere un doppio riferimento negli oggetti;

navigazione dei dati : nel modello ad oggetti, è possibile navigare nel grafo degli oggetti utilizzando i riferimenti: non esiste però una traslazione diretta di questo comportamento nel modello relazionale, e ciò si traduce in un accesso spaventosamente inefficiente ai dati, in quanto sarebbe preferibile minimizzare il numero di query SQL generate e sfruttare i costrutti di JOIN per esplorare le associazioni.

1.2.3 Hibernate

Hibernate è una piattaforma middleware open source per la gestione della persistenza relazionale, per Java, che ad oggi riscuote un buon seguito tra i programmatori. Il progetto è nato nel 2001, venendo successivamente assorbito e supportato dalla JBoss Community. Ad oggi Hibernate può definirsi un progetto consolidato, che ha raggiunto un'ottima stabilità e una buona penetrazione nel mercato, tanto da potersi autodefinire uno standard *de facto*, continuando ad aggiornarsi per rispecchiare l'introduzione dei nuovi standard Java.

1.2.4 Eclipse_Link

Al pari di Hibernate, Eclipse_Link è un framework open source per la gestione della persistenza, basato su standard consolidati. In questo caso il progetto viene sviluppato sotto l'egida di Eclipse, quindi è facilmente interfacciabile con l'Eclipse IDE⁵ per facilitare lo sviluppo di applicazioni.

Rispetto a Hibernate soffre certamente una minore penetrazione nel mercato e una minore offerta di funzionalità, ma per applicazioni di piccola dimensioni è comunque una alternativa praticabile.

1.2.5 JPA

Entrambi gli ORM che andremo a testare prevedono diverse tecnologie, e quindi diverse sintassi, per l'effettiva traduzione tra la logica ad oggetti e il modello relazionale. L'alternativa che noi andremo ad analizzare e testare durante i test su persistenza, in accoppiamento agli ORM, è JPA (JAVA PERSISTENCE API).

⁵Un INTEGRATED DEVELOPMENT ENVIRONMENT è un tool di sviluppo software, che integra le funzioni di editor con quelle di compilatore e di debugger, e può prevedere vari strumenti di autocomposizione o di visualizzazione della struttura del codice.

JPA è stato introdotto a partire dalla release 5 di Java, con lo scopo di potenziare, ma allo stesso tempo rendere più sfruttabile per lo sviluppatore, i vari tentativi di interfaccia tra oggetti del linguaggio e database relazionali che erano stati proposti nel corso degli anni. A tutti gli effetti JPA può essere definito come uno strumento di ORM, in quanto rende possibile una mappa diretta tra oggetti creati all'interno del linguaggio Java, e la persistenza in database relazionali.

Dal punto di vista del linguaggio, è sufficiente includere nella definizione delle classi da gestire una serie di annotazioni, definite dallo standard, che si occupano appunto di descrivere il comportamento dell'oggetto nei confronti della persistenza (quali campi memorizzare, il tipo di caricamento, la presenza di chiavi esterne o di relazioni tra i dati).

Per l'effettiva gestione della persistenza, JPA si appoggia a uno strato software che viene definito come *provider*, che si occupa appunto di tradurre la sintassi JPA in funzioni del modello relazionale, solitamente compilando delle query SQL.

E' importante notare come sia Hibernate, sia Eclipse_Link possano essere utilizzati come dei *provider* in quest'ottica, e come sia possibile sfruttare un'unica sintassi JPA, modificando il *provider* utilizzato con delle variazioni minime e localizzate nel codice.

1.3 Apache Tomcat

Apache Tomcat (o semplicemente Tomcat) è una implementazione open-source delle specifiche relative a Java Servlet (vedi sezione 1.3.1) e JavaServer Pages (vedi sezione 1.3.2). Svolge quindi la funzione di contenitore per applicazioni Web, integrando sia le funzioni di base di un web-server, sia la possibilità di gestire contenuti dinamici che vengono implementati nel linguaggio Java.

In sostanza, Tomcat si occupa di fornire un'interfaccia a un generico *web agent*, che richieda una risorsa web: se si tratta di una semplice risorsa statica, Tomcat si limita a determinare la risposta comportandosi come un comune web server.

Viceversa, se si tratta di una risorsa dinamica, assume il ruolo di contenitore, ossia determina il tipo di applicazione dinamica da attivare (sia essa una Servlet o una JSP), e ne gestisce l'intero ciclo di vita sino alla generazione della risposta e al trasferimento al richiedente.

1.3.1 Java Servlet

Le Servlet sono delle classi, scritte in linguaggio Java, che rispettano le *specifiche* previste dalla Java Servlet API. In questo modo, l'accesso ad una servlet viene regolamentato, garantendo che tali componenti possano inte-



Figura 1.2: Logo di Apache Tomcat [7]

grarsi in modo semplice con il *web container* che le ospita (e che nel nostro caso potrà essere Tomcat, oppure la piattaforma App Engine).

Grazie alla standardizzazione delle specifiche, e all'uso del linguaggio Java, è possibile creare applicazioni Web che siano indipendenti sia dal server, sia dalla piattaforma, e allo stesso modo possano sfruttare l'intero set di strumenti che il linguaggio Java mette a disposizione, come ad esempio le librerie JDBC per l'accesso ai database.

L'utilizzo principale delle servlet, che sarà anche utilizzato dai nostri test, prevede l'utilizzo delle Servlet in risposta ai metodi previsti dal protocollo HTTP⁶: è comune in questo caso parlare di HTTP Servlet.

1.3.2 JavaServer Pages

Analogamente alle servlet, la tecnologia JSP consente di specificare dei contenuti web dinamici tramite il linguaggio Java, differenziandosi nell'approccio seguito: invece di realizzare delle classi interamente specificate dal linguaggio Java, che hanno un proprio ciclo di vita e sono attivate dal web container, in questo caso si creano delle pagine ibride in cui convivono contenuti statici e dinamici.

Quando viene richiesto il caricamento di una pagina JSP, il ruolo principe viene svolto dal motore JSP, che si occupa appunto di elaborare il documento JSP richiesto.

Le parti statiche della pagina, che possono contenere ad esempio del codice HTML o XML, sono trasferite in modo diretto come output, mentre il codice Java delle sezioni dinamiche viene eseguito, e l'output generato andrà a concorrere alla composizione del risultato.

In questo modo la composizione di componenti dinamici segue un approccio certamente più immediato e meno strutturato, a fronte di una minore potenza espressiva. Nella maggioranza dei casi, un web container comprende anche un motore JSP: nel nostro caso, sia Tomcat sia App Engi-

⁶L'**HYPertext TRansfer PROTOCOL** è un protocollo di rete di livello applicazione inizialmente pensato per il trasferimento di ipertesti che è stato esteso in modo più generico e *stateless* per altri scopi, tra i quali i web services [13].

ne consentono l'utilizzo di questa tecnologia, anche se la nostra trattazione non ne farà uso.

2 GOOGLE APP ENGINE

Presenteremo ora una trattazione dettagliata di Google App Engine, la piattaforma di cloud computing sviluppata da Google.

Tale piattaforma consente lo sviluppo di applicazioni web nei linguaggi Java, Python e Go, tramite una suite di servizi che si propongono di supportare la scalabilità anche sotto elevati carichi, in modo trasparente all'utente.

2.1 Architettura di App Engine

In questa sezione andremo a descrivere l'architettura della piattaforma, e i meccanismi che vengono attivati in relazione alle richieste web. Come approfondimento si rimanda al testo cartaceo, oppure alla documentazione ufficiale on-line [12].

Ad alto livello, App Engine può essere suddiviso in tre parti logicamente distinte, che cooperano al fine di realizzare un servizio di hosting per applicazioni web. Possiamo quindi riconoscere

Il Runtime Environment un'applicazione di App Engine viene avviata in risposta ad una *web request*. La web request viene generata in seguito ad una HTTP request generata da un client (nella maggioranza dei casi, il client è un browser che rappresenta l'interfaccia con l'utente, ma vedremo in seguito come siano possibili altre modalità).

L'ambiente di runtime raccoglie i meccanismi che consentono la gestione delle web request, a partire dall'individuazione dell'applicazione da attivare fino alla consegna del risultato come *HTTP response*. L'impostazione dell'ambiente specifica anche una serie di vincoli all'applicazione, relativi sia alla sicurezza che alla gestione delle performance. Per maggiori dettagli si rimanda alla sezione 2.2.

Il datastore: l'ambiente di runtime tratta ciascuna web request come indipendente dalle altre, e non assicura una persistenza dei dati tra richieste successive, essenzialmente per massimizzare la scalabilità. Il datastore si occupa invece di fornire una base di persistenza a lungo termine: anche in questo caso, la scalabilità condiziona pesantemente l'organizzazione progettuale, tanto che non si ha a che fare con un classico DBMS relazionale, bensì con un database distribuito di *entities*, che comporta una serie importante di vincoli. Il datastore è trattato in modo esteso nella sezione 2.3.

I servizi complementari: si tratta di una serie di API che consentono alle applicazioni di interagire ad alto livello con altre entità (tramite

fetch di URL, invio e ricezione di mail e messaggi XMPP¹, semplici trasformazioni su immagini). Un ulteriore servizio è la memcache, che gestisce una veloce persistenza a breve termine, e consente il caching di entità per minimizzare gli accessi al datastore che potrebbero rappresentare un rallentamento dell'applicativo.

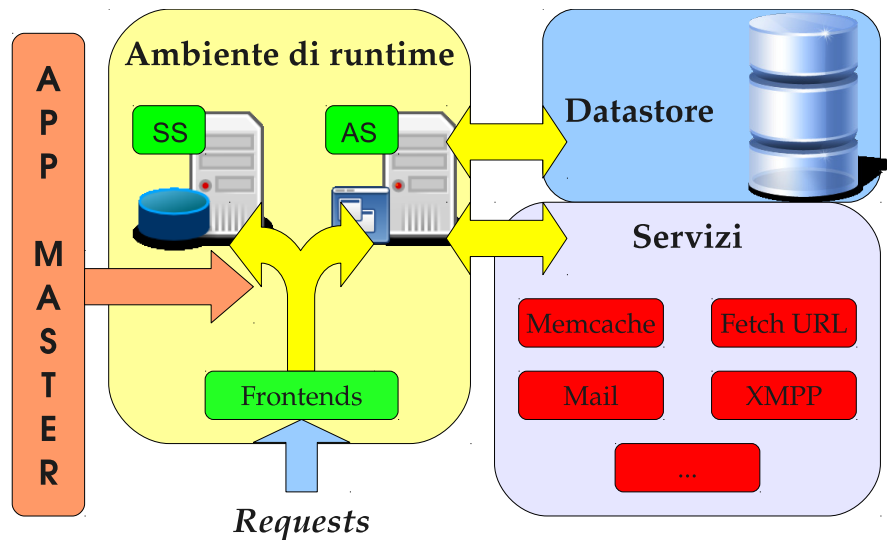


Figura 2.1: Schema dell'architettura di App Engine

Come sarà evidenziato più volte, l'obiettivo principe dell'infrastruttura è l'elevata scalabilità, intesa come la *capacità di non subire un degrado di prestazioni in relazione all'accesso concorrente di un elevato numero di utenti*.

Tale scelta influenza pesantemente sia l'architettura del prodotto, sia i vincoli imposti al programmatore di web application: App Engine si pone dunque come una efficiente piattaforma per applicazioni dinamiche real-time, a ridotto tempo di risposta e ad elevato carico. In questo senso, sacrifica la semplicità d'uso e la fruibilità come web-hosting general purpose: vedremo come sia comunque possibile utilizzarlo sia come base per siti di tipo tradizionale, sia per servizi che richiedono una elevata integrazione tra i dati (ad esempio di tipo gestionale), ma in questo caso si deve scendere a compromessi e accettare una complicazione, a volte molto invasiva, nella realizzazione del software.

¹L'EXTENSIBLE MESSAGING AND PRESENCE PROTOCOL è un protocollo basato su XML per lo scambio di informazioni strutturate in quasi real-time tra due nodi della rete [14] [15].

2.2 Runtime environment

L'ambiente di runtime è la parte dell'architettura App Engine designata a rispondere alle web request: ricordiamo come tutte le logiche in questa sezione debbano essere intese come *stateless*, nel senso che ogni richiesta viene trattata in modo indipendente dalle altre.

2.2.1 Frontends

Il frontend si occupa di ricevere le *web request* indirizzate ad App Engine, attivando gli altri strumenti del ambiente di runtime e attendendo le risposte, dopodichè si occupa di instradare queste risposte al client. All'arrivo di una nuova request, il frontend:

determina l'*app id* dell'applicazione da attivare, in base al sotto-dominio dell'URL presente nella richiesta;

carica i files di configurazione relativi all'applicazione richiesta, che specificano ad esempio il mapping delle servlet, gli handler registrati e il tipo di risorse da produrre;

determina la risorsa richiesta in base all'URL della richiesta;

determina il tipo di risorsa in base all'estensione contenuta nell'URL e alla configurazione precedentemente caricata; le risorse possono essere:

files statici : non richiedono nessun processing, sono immutabili tra una richiesta e l'altra e devono essere semplicemente trasmessi al client (ad esempio i files contenenti immagini); sono memorizzati nei server di file statici (vedi 2.2.2);

request handler : sono risorse che richiedono un processing per generare una risposta, come una servlet o una pagina JSP, e vengono eseguiti all'interno degli application server.

resituisce la risorsa al client, gestendo la connessione e eventuali parametri opzionali come la compressione.

Osserviamo che, a differenza dei server di files statici dove è sufficiente richiedere la ricerca alla macchina opportuna, il ricorso ad un request handler rende necessario un meccanismo di *load balancing* che rientra tra le prerogative del frontend. In fase di inoltro della richiesta all'application server, il frontend determina su quale server instradare la richiesta in base a una stima dei carichi operativi e del pattern di utilizzo delle risorse.

2.2.2 Static file servers

I server di file statici si occupano della memorizzazione dei files statici, ossia dei files che possono essere trasmessi al client senza la necessità di una elaborazione: in questo modo, si evita di occupare risorse computazionali negli application server per recuperare files che devono essere restituiti pedissequamente.

Tali server hanno una architettura ottimizzata per la funzione che devono svolgere, ad esempio richiedono meno potenza di calcolo, ma spazio di memorizzazione più capiente e a latenza minore. Allo stesso tempo, la topologia della rete che li collega al frontend è ottimizzata al fine di ridurre i tempi di trasferimento.

2.2.3 App servers

I request handler, al contrario dei file statici, richiedono una sezione di processing per generare la risorsa. Di conseguenza la loro impostazione architetturale privilegerà la potenza di calcolo e la memoria disponibile. Per migliorare i tempi di caricamento delle app, in questi server l'ambiente di runtime (inteso in questo caso come l'interprete del linguaggio e il set di librerie di sistema) è sempre residente in memoria.

In App Engine è possibile specificare i request handler sia in Java (di conseguenza su un set di linguaggi per i quali è disponibile un interprete all'interno della JVM, come PHP e Ruby), sia nel linguaggio Python; la trattazione che segue si focalizza unicamente sul linguaggio Java, in quanto il set di funzionalità offerte sulle due diverse piattaforme software è in massima parte coincidente.

L'allocazione della CPU tende a privilegiare le applicazioni estremamente veloci, in generale si garantisce comunque una alta priorità nello scheduling del processore fintanto che il tempo di risposta non supera il secondo. Al di sopra di questa soglia, lo scheduler provvede progressivamente a ridurre l'accesso alle risorse di calcolo in modo da mantenere elevato il throughput.

Le applicazioni vengono eseguite all'interno di una *sandbox*: in questo modo si realizzano una serie di vincoli che eliminano alcune problematiche di scalabilità, ad esempio non è possibile generare connessioni su socket esterni, suddividere il lavoro attraverso la creazione di nuovi processi o thread, accedere al file system locale in scrittura o eseguire codice nativo attraverso JNI².

Allo stesso tempo, la *sandbox* garantisce che l'esecuzione di due applicazioni dello stesso tipo sulla stessa macchina non possa generare proble-

²JAVA NATIVE INTERFACE è una interfaccia di programmazione che consente l'esecuzione di codice nativo, ossia specifico per la piattaforma software della macchina in uso, e consente quindi l'interfaccia con codice scritto in linguaggi di programmazione diversi.

matiche di concorrenza: in questo senso si isola il più possibile la singola app, in modo da garantire la massima scalabilità.

Una particolare menzione merita l'*app caching*, ossia il meccanismo che genera l'allocazione delle risorse quando viene caricata una nuova app. Quando viene richiesta l'esecuzione di una applicazione per la prima volta (o quando la stessa non è in memoria in quanto è uscita dalla cache) il primo passo è il caricamento dell'*app environment*, al quale segue il caricamento di tutte le classi necessarie all'esecuzione della servlet richiesta.

Il compito del meccanismo di caching è quello di mantenere nella memoria di lavoro le applicazioni in modo da ridurre i tempi di caricamento per le richieste successive, assicurando performance accettabili soprattutto dal punto di vista della scalabilità.

Nel nostro caso, gli application server mantengono in memoria anche le classi importate e le sezioni marcate come *static* delle classi utilizzate, che in caso di richieste successive eviteranno quindi di essere ricaricate.

Occorre tenere ben presente questo comportamento, in quanto è consigliabile sfruttarne i meccanismi per migliorare le prestazioni dei propri applicativi, ad esempio caricando strutture dati riutilizzabili nelle sezioni *static*, come quelle generate dal parsing dei file di configurazione.

Allo stesso tempo, bisogna ricordare che è possibile che all'interno dello stesso *app environment* siano in esecuzione più istanze di una stessa applicazione che vengono gestiti da App Engine all'interno di thread concorrenti: tali istanze condividono però l'accesso alle sezioni in *app caching*, ne consegue la richiesta di un comportamento *thread safe* per evitare interferenze tra app concorrenti.

2.2.4 App master

L'app master si pone al di sopra dei componenti appena descritti, ed è responsabile del loro controllo: una delle sue principali funzioni è ad esempio la gestione delle versioni delle applicazioni in uso, comprendendo sia il deploying sia la definizione della versione da caricare in risposta alle *web request*.

2.3 Datastore

Relativamente al Runtime Environment, abbiamo evidenziato come la gestione delle web request segua una logica *stateless*, dove il risultato dell'elaborazione viene determinato unicamente dagli attributi della *web request*: per introdurre una logica *stateful* è necessario disporre di uno strumento che garantisca la persistenza dei dati attraverso le diverse chiamate.

App Engine propone una persistenza basilare sfruttando la Memcache, che definisce una persistenza in memoria ad accesso veloce, ma con garanzie di affidabilità e transazionalità molto ridotte. Descriviamo ora il Da-

tastore, dove la persistenza viene realizzata su memoria di massa secondo un meccanismo distribuito, offrendo garanzie di consistenza e affidabilità molto più elevate, al prezzo di una maggiore lentezza nell'esecuzione delle operazioni di accesso e modifica.

La maggior parte degli sviluppatori può vantare una certa esperienza con i database relazionali, che ad oggi rappresentano l'opzione più praticata per quanto riguarda la costruzione di una base di dati. App Engine non mette però a disposizione dei propri utenti un DBMS relazionale, ma una base di dati distribuita che si differenzia nettamente dal modello più in voga, tanto da essere denominata *Datastore* e non *database* come ci si sarebbe potuti aspettare.

La ragione di tale scelta deve essere ricercata ancora una volta nella ricerca della massima scalabilità:

- Google disponeva di una piattaforma di persistenza ad elevata scalabilità, basata sulla tecnologia *BigTable* (vedi sezione 2.3.1) sviluppata internamente e utilizzata a supporto dei propri servizi, come la *web search*, GMail, Google Maps e altri, ampiamente collaudata;
- tale piattaforma richiedeva quindi un adattamento software minimale per la costruzione dell'interfaccia di App Engine;
- al contrario, la configurazione di un DBMS tradizionale al fine di garantire un ottimo compromesso tra scalabilità e affidabilità comporta una complicazione non banale, in quanto richiede l'introduzione di meccanismi di replicazione e load balancing che devono essere attentamente configurati.

E' stata quindi privilegiata la semplicità architetturale, mettendo a disposizione dell'utente la stessa tecnologia di persistenza che veniva già utilizzata internamente. Questo comporta però una maggiore difficoltà allo sviluppatore medio, che in molti casi dovrà rimodulare il proprio approccio nella gestione della persistenza.

Allo stesso tempo, il Datastore introduce una serie di vincoli sulle possibili operazioni eseguibili sui dati, che non contemplano alcune caratteristiche che per l'utente del modello relazionale sono considerate ormai naturali (ad esempio la possibilità di utilizzare operazioni di JOIN senza particolari restrizioni).

Per venire incontro alle esigenze di questi utenti, è stata annunciata l'introduzione di un servizio di *Hosted SQL*[\[11\]](#) come alternativa all'utilizzo del Datastore, che dovrebbe offrire un DBMS dedicato con pieno supporto a SQL. L'introduzione di tale opzione viene annunciata a oggi come plausibile in tempi medio-brevi: è però molto difficile dare una panoramica di tale opzione, in quanto si preannuncia che l'uso di tale servizio comporterà un costo aggiuntivo per l'utente, al momento non ancora specificato. Allo

stesso tempo non sono ancora stati resi noti le funzionalità e le caratteristiche del servizio che verrà implementato, ad esempio per quanto riguarda le garanzie di scalabilità e l'affidabilità.

A Gennaio 2011 è stata introdotta una variante del Datastore denominata *High Replication*, che si differenzia dalla configurazione iniziale denominata *Master/Slave* nell'organizzazione della replicazione dei dati [16]. E' possibile scegliere una delle due varianti per l'esecuzione della propria applicazione, tenendo conto che la nuova variante garantisce una maggiore affidabilità nell'accesso ai dati, ottenuta modificando il modello di replicazione dei dati.

Nella versione *Master/Slave* la scrittura di un dato comporta l'accesso ad un singolo data center denominato *master*, dopodichè la replicazione avviene in modo asincrono su altri datacenter denominati *slave*: in questo modo la richiesta di risorse è minima e si privilegiano le performance, ma esiste la possibilità di una perdita o di una inutilizzabilità temporanea dei dati, in caso di inconvenienti sulla piattaforma.

Nella versione *High Replication* la scrittura viene replicata in modo sincrono utilizzando l'algoritmo di Paxos [17] per la gestione del consenso, al costo di latenze maggiori e un maggior uso delle risorse (circa triplo), che si rifletterà in un maggior costo per l'esecuzione dell'applicazione.

Gli oggetti che vengono memorizzati nel Datastore assumono il nome di *entity*: presenteremo una panoramica di come questi oggetti vengono effettivamente memorizzati e di come è possibile accedervi successivamente, per maggiori dettagli si rimanda alla sezione del Datastore sulla guida on-line [12].

2.3.1 Bigtable

Il Datastore di App Engine sfrutta la tecnologia BigTable per la gestione della memoria di massa. Presentiamo quindi una veloce panoramica di tale tecnologia, basandoci su [18], cui si rimanda per approfondimenti.

Bigtable è un sistema di memorizzazione distribuito per dati strutturati, costruito per la scalabilità su grande dimensione (centinaia di petabyte³ di dati distribuiti su centinaia di server), ed è correntemente utilizzato da gran parte dei servizi Google tra cui il *web indexing* e Google Earth.

E' interessante notare come tali servizi richiedono un altro grado di flessibilità alle infrastrutture, in quanto sia il carico di lavoro sia le dimensioni dei dati hanno una variabilità molto marcata, potendo passare da URL di pochi byte a immagini satellitari di grandi dimensioni, che possono richiedere elaborazioni in backend a elevato throughput oppure risposte dirette all'utente con latenza limitata.

³10¹⁵ byte, equivale a 1000 terabyte

Tale sistema non prevede un supporto completo al modello relazionale, in quanto si concentra su un modello più semplice che consente al client la padronanza del formato e del layout dei dati, oltre a una forma di controllo sulla località dei dati all'interno della topologia di storage.

Bigtable viene definito come una *mappa multi-dimensionale ordinata, sparsa, distribuita e persistente*: tale mappa viene indicizzata attraverso tre attributi, ossia una chiave di riga, una di colonna, e un timestamp, che fanno riferimento ad un array di byte non interpretato.

Una *chiave di riga* è una stringa arbitraria: i dati vengono mantenuti in ordinamento lessicografico in base a tale indice. E' interessante notare come ciascuna operazione di accesso sia atomica a livello di riga (indipendentemente dal numero di colonne coinvolte). Le righe appartenenti a una stessa tabella vengono dinamicamente partizionate in *tablets*, che sono le unità di lavoro per il partizionamento e il load balancing: in questo modo è possibile garantire la località sia per colonne all'interno della stessa riga, sia per righe lessicograficamente adiacenti.

Le *chiavi di colonna* sono raggruppate in insiemi chiamati *column families*, che formano l'unità base per l'accesso: solitamente tutti i dati memorizzati all'interno di una famiglia sono dello stesso tipo, in quanto è prevista una compressione a livello di famiglia. Una chiave di colonna è costruita dalla sintassi *family:qualifier*, dove l'identificatore di famiglia deve essere una stringa di caratteri stampabili mentre il qualificatore può essere una stringa arbitraria.

Il *timestamp* consente di mantenere diverse versioni dello stesso dato, che sono memorizzate per timestamp decrescente in modo da accedere alle versioni più recenti per prime: sono disponibili delle funzioni di *garbage collection*, che consentono di mantenere vive solamente le *n* versioni più recenti, oppure di definire una età massima per la sopravvivenza del dato.

Il *dato indicizzato* dai tre attributi è gestito come una stringa binaria: sta al client che utilizza questa infrastruttura la sua interpretazione in un tipo di dato primitivo oppure la gestione della serializzazione di strutture più complesse.

Infine, è interessante osservare come BigTable sfrutti GFS (*Google File System*, si veda [19]) per memorizzare sia i dati sia i files di log: GFS è un file system distribuito, ad elevata scalabilità, che mette a disposizione un accesso alla memorizzazione su disco con elevati standard di performance e fault-tolerance, pur utilizzando hardware di fascia media.

2.3.2 Entities & keys

Come precedentemente accennato, ciascun oggetto gestito dal Datastore assume la denominazione di *entity*: ogni entità è univocamente identificata da un *kind* (il tipo di entità) e da un *entity ID* (ossia un identificatore, univo-

co all'interno delle entità dello stesso tipo, che può essere deciso dall'utente o generato automaticamente dal sistema).

Ciascuna entità può possedere delle *properties*, ossia degli attributi che possono appartenere a un qualche tipo di dato come stringhe, numeri interi, date, variabili binarie. A differenza delle tabelle del modello relazionale, il Datastore non richiede che tutte le entità appartenenti allo stesso tipo definiscano lo stesso insieme di proprietà.

Inoltre, una proprietà può contenere uno o più valori: nel caso in cui utilizzi valori multipli, ciascuno di essi può appartenere ad un tipo di dato differente.

2.3.3 Query e indici

L'accesso alle entità memorizzate può essere realizzato attraverso le query: una query può operare su oggetti appartenenti ad un singolo *kind*, specificando una lista di filtri (condizioni che devono essere realizzate dalle proprietà dell'entità affinché essa sia inclusa nella lista dei risultati) e delle condizioni di ordinamento in cui presentare i risultati.

Ciascuna query si basa sull'accesso ad un indice: in questo modo, la verifica della corrispondenza ai filtri richiede un semplice accesso ai filtri, dopodiché le entità vengono estratte sequenzialmente dalla tabella ripercorrendo la sequenza delle chiavi all'interno dell'indice.

In questo modo, la complessità computazionale della query è correlata unicamente alla taglia del *result set* e non al numero di entità appartenenti al *kind*, per garantire la stabilità. Secondo questa definizione, una query che estrae 100 entità, dovrebbe avere prestazioni idealmente costanti se eseguita su un datastore contenente 200, 20000 o 2000000 di oggetti.

E' importante notare come la definizione degli indici venga in massima parte svolta dall' App master, in quanto in fase di deploying dell'applicazione le possibili query generate vengono analizzate e per ognuna di esse vengono generati gli indici necessari; è però possibile specificare manualmente degli indici supplementari attraverso gli strumenti di amministrazione.

2.3.4 Transazioni e *entity groups*

Il concetto di transazione è familiare agli sviluppatori che già operano sui DBMS tradizionali: App Engine mette a disposizione questo utile costrutto anche per il suo datastore, ponendo tuttavia una serie di limitazioni che come vedremo sono direttamente correlate con la struttura della piattaforma Bigtable sottostante.

Nella documentazione, App Engine definisce una transazione come un insieme di operazioni sul Datastore, essenzialmente basandosi sul concetto di *atomicità*: per definizione, una transazione non è mai applicata in

modo parziale, quindi le operazioni che la compongono vengono eseguite nella loro interezza in caso di successo, altrimenti nessuna di queste viene applicata.

Le cause di fallimento di una transazione sono molteplici, e possono spaziare da errori interni del Datastore, a modifiche concorrenti sulle stesse entità che devono essere regolamentate, al raggiungimento della quota impostata per una qualche risorsa. In questi casi l'errore viene segnalato da una apposita eccezione, anche se può capitare che una transazione possa comunque essere portata a termine anche in presenza di una eccezione.

Per poter usufruire delle transazioni, è necessario applicare in modo preciso il concetto di *entity group*, in quanto solo entità appartenenti allo stesso gruppo possono essere modificate all'interno di una singola transazione. Dal punto di vista della localizzazione all'interno del Datastore, le entità appartenenti ad un gruppo sono memorizzate in modo contiguo nell'infrastruttura di BigTable, in modo da ricercare latenze minori e da garantire la possibilità di modifiche atomiche senza una complessità troppo elevata nella gestione della sincronizzazione.

Le applicazioni definiscono i gruppi basandosi sul concetto di *parent entity*. Un'entità che non definisce nessun *parent* viene detta *root entity*, e costituisce quindi un nuovo gruppo. Successivamente, tutte le entità che specificano una *parent entity* appartengono allo stesso gruppo di quest'ultima.

Occorre notare come un'entità che è *parent* possa a sua volta specificare una propria *parent*, in modo gerarchico, fino alla creazione di un *path* di *ancestor* che fa capo ad una unica *root*. La definizione della struttura gerarchica assume quindi una grande importanza della creazione di una App, al fine di trovare il giusto compromesso tra la possibilità di definire transazioni e le performance, ricordando che le modifiche concorrenti su gruppi diversi sono potenzialmente molto più performanti.

La persistenza della transazione viene ovviamente garantita, salvo seri inconvenienti che compromettano la piattaforma del Datastore, in quanto gli effetti di una transazione che ha avuto successo non corrono rischi di essere persi: è possibile incrementare il vincolo di persistenza sfruttando l'High Replication Datastore.

Per quanto riguarda l'isolamento, viene garantito il livello massimo, ossia *SERIALIZABLE*, che mette al riparo da tutti i possibili errori indotti dall'accesso concorrente ai dati. Si noti come, non utilizzando le transazioni, il livello garantito sia pari a *READ_COMMITTED* nell'accesso alle entità, mentre sia minore nell'interazione tra query e entità a causa di un aggiornamento degli indici ritardato rispetto alla modifica dell'entità.

Passando infine alla consistenza, si garantisce che tutte le query e gli accessi alle entità durante una transazione vedono un unico *snapshot* consistente del datastore, pari a quello esistente all'inizio della transazione: a differenza del comportamento presente nella maggioranza dei database,

gli accessi all'interno della transazione *NON* sono in grado di vedere i risultati di modifiche precedenti alle entità avvenute all'interno della stessa transazione.

2.3.5 API di accesso

Come abbiamo visto, il Datastore è stato costruito sulla piattaforma BigTable preesistente, definendo delle logiche ad un livello di astrazione maggiore (entità invece di singoli valori all'interno della mappa distribuita), ma allo stesso tempo disciplinando gli accessi e le operazioni permesse in modo da rispettare i vincoli architetturali e da incentivare l'accesso concorrente a scalabilità elevata.

L'interfaccia software tra questa struttura logica e il linguaggio con cui viene realizzata l'applicazione (che nel nostro caso è Java) comprende ben 4 soluzioni alternative, in modo da semplificare l'apprendimento agli sviluppatori che hanno già una conoscenza di una tipologia di accesso ai DBMS tradizionali, oltre a tentare di semplificare il porting di applicazioni preesistenti, sebbene con molte limitazioni.

2.4 Preparazione dell'ambiente di lavoro

Come primo passo spiegheremo come organizzare l'ambiente di lavoro per la creazione e il deployment di applicazioni su App Engine. Tale sezione può rappresentare un'utile guida per il lettore che voglia predisporre la propria macchina e iniziare a sviluppare su questa piattaforma.

2.4.1 Registrazione di un Google Account

Per lo sviluppo su App Engine occorre disporre di un account Google, che deve essere successivamente abilitato alla creazione di applicazioni. Si è scelto di creare un account ad hoc (*tesi.conte@gmail.com*) per i test relativi all'argomento di tesi, pertanto tutti i passaggi successivi utilizzeranno tale account come base per le spiegazioni. Tale account è stato inoltre condiviso con il gruppo di lavoro del laboratorio Sintesi.

Iniziamo creando il nostro account Gmail, alla pagina <https://www.google.com/accounts/NewAccount?service=mail>.

A questo punto disponiamo di un Google account standard, che ci consente di usufruire della suite di prodotti Google e, eventualmente, anche di accedere ad applicazioni su App Engine già funzionanti che utilizzino tali account.

Il passo successivo consiste nell'abilitare il nostro account alla creazione di applicazioni di App Engine. Basterà accedere alla pagina <https://app-engine.google.com>: una volta entrati, il nostro primo tentativo di creare

una applicazione comporterà la richiesta del sistema di verificare l'account tramite un sms.

Si noti come tale procedura sia richiesta solo all'atto della creazione della prima applicazione, e serva per associare all'account almeno un riferimento fisico (in questo caso un numero di cellulare). Si ha quindi il vincolo che una determinata utenza telefonica potrà abilitare un solo account.

2.4.2 Installazione di Eclipse IDE

Coerentemente con quanto consigliato dal testo, scegliamo di utilizzare Eclipse IDE come strumento per lo sviluppo del codice. Questa scelta viene motivata dalla notevole maturità e diffusione del prodotto, e dalla possibilità di installare un apposito plugin (si veda alla sottosezione successiva) che semplifica enormemente il deploying.

Si ripercorrerà brevemente il percorso di installazione del prodotto: nel nostro caso tale software è stato installato su una macchina che monta il sistema operativo Ubuntu Linux, quindi certe sezioni risulteranno specifiche per tale sistema operativo e dovranno essere adattate su sistemi operativi differenti.

Il primo passo è ovviamente il download del software, dal sito <http://www.eclipse.org/downloads/>: occorre scegliere la versione corrispondente al sistema operativo che si sta utilizzando, dopodiché si avvierà il download dell'archivio compresso. In generale tale pacchetto andrà decompresso in qualche directory del file system locale, e si dovrà creare un qualche tipo di collegamento all'eseguibile che lancia l'applicazione.

Sono disponibili molti pacchetti diversi, che si differenziano per i linguaggi supportati e i plugin compresi: il pacchetto che fa al caso nostro è l' *Eclipse IDE for Java EE Developers*: ricordiamo che future estensioni possono essere installate direttamente dal menu di Eclipse, passando per *Help -> Install new software...*

Al momento la versione stabile più recente è la Helios (3.6), il numero di versione è importante in quanto il plugin da aggiungere è strettamente correlato a tale indicatore.

Limitatamente alla distribuzione Ubuntu Linux, una buona soluzione per installare il prodotto potrebbe essere la seguente, basata su [10]:

1. Aprire una finestra di terminale
2. Create la cartella /opt se non è già esistente – `sudo mkdir /opt`
3. Portarsi nella cartella in cui il browser ha salvato il download di Eclipse – `cd Scaricati`
4. Decomprimere il pacchetto appena scaricato – `tar -xvf eclipse-jee-helios-SR2-linux-gtk-x86_64.tar.gz`

5. Spostare la cartella decompressa nella cartella /opt – `sudo mv eclipse /opt`
6. Creare una cartella bin che conterrà il file di avvio – `sudo mkdir /opt/bin`
7. Creare un file di testo – `sudo gedit /opt/bin/eclipse` – e compilarlo come segue

```
export MOZILLA_FIVE_HOME=/usr/lib/mozilla/  
export ECLIPSE_HOME=/opt/eclipse  
$ECLIPSE_HOME/eclipse $*
```
8. Rendere eseguibile il file appena creato `sudo chmod +x /opt/bin/eclipse`
9. Aggiungere un *lanciatore* al pannello superiore del desktop, con click destro del mouse che aprirà il menu da cui scegliere Aggiungi al pannello -> Lanciatore applicazione personalizzato, che andrà impostato come segue
Tipo: Applicazione
Nome: Eclipse Platform
Comando: `gksudo /opt/bin/eclipse`
Icona: `/opt/eclipse/icon.xpm`

A questo punto, cliccando sul lanciatore che abbiamo appena creato dovremmo poter aprire Eclipse, pronta alla creazione di progetti Java e con varie funzionalità relative alla versione EE, ma non ancora pronto per creare progetti App Engine.

2.4.3 Configurazione di Eclipse IDE

Passiamo quindi a installare il plugin, fornito da Google, che consente di velocizzare enormemente la gestione di un progetto App Engine. Passiamo dunque dal menu Help -> Install new software -> Add, nella finestra che si aprirà inseriremo

Name: Plugin per App Engine

Location: <http://dl.google.com/eclipse/plugin/3.6>.

A questo punto nella lista degli strumenti disponibili compariranno i due gruppi Plugin e SDKs, è sufficiente selezionarli entrambi e confermare i passaggi successivi per avere il attivare il plugin.

3 TEST FUNZIONALI SUL DATASTORE APP ENGINE

Dopo aver presentato l'architettura e le funzionalità salienti di Google App Engine, soffermandoci in particolare sul datastore, andremo ora a realizzare e documentare una serie di test funzionali.

Tali test hanno lo scopo di mettere in luce la difficoltà implementativa del codice, il supporto di funzionalità che ad oggi si ritengono basilari nello sviluppo di applicazioni web, e la possibilità di riutilizzare codice scritto per applicazioni su web container tradizionali.

3.1 Persistenza di una singola entità

3.1.1 Descrizione

Lo scopo di questo test è di verificare la riutilizzabilità del codice, nella migrazione verso un progetto App Engine a partire da un progetto JPA (per una descrizione delle specifiche JPA, si veda la sezione 1.2.5).

In questo senso si andrà a verificare la mole di lavoro che comporterebbe adattare un progetto già esistente, sviluppato secondo le specifiche JPA, e la possibilità che la migrazione del progetto verso App Engine vada a inficiare la compatibilità con l'ORM precedentemente utilizzato.

Si tratta di un semplice accesso CRUD¹ ad una tabella di un database PostgreSQL, tramite due provider JPA come EclipseLink e Hibernate (per dettagli fare riferimento alle sezioni 1.2.4 e 1.2.3), che viene successivamente modificato in un progetto App Engine, dove il nuovo provider JPA sarà proprio il DataNucleus² che si interfaccia con le BigTables di App Engine.

3.1.2 Creazione del progetto

Il primo passo è la creazione di un progetto su Eclipse. E' sufficiente creare un *JPA Project*, specificarne il nome che nel nostro caso è *test-crud-temp*, e lasciare le impostazioni di default nei primi due pannelli.

Nel terzo pannello, relativo alle *User Libraries*, è necessario settare la *Platform* a *EclipseLink 2.1.x*, inoltre occorrerà caricare le librerie esterne. Occorre innanzitutto scaricare quattro pacchetti:

¹CRUD è un acronimo che indica le 4 funzioni basilari che caratterizzano l'accesso alla persistenza, ossia **C**REATE - **R**ETRIEVE - **U**PDATE - **D**ESTROY e rappresenta un interessante banco di prova per vedere come JPA modella queste operazioni logiche in metodi implementati

²Si noti che DataNucleus potrebbe essere utilizzato come JPA Provider anche per l'accesso al database PostgreSQL: al momento si tralascia tale possibilità in quanto si vogliono stimare le difficoltà nel porting a partire da scenari di ORM consolidati

Il pacchetto completo di **EclipseLink** che contiene i file jar che ci servono oltre alla documentazione e altre utilities, nel nostro caso nella versione 2.2.0, da <http://www.eclipse.org/downloads/download.php?file=/rt/eclipselink/releases/2.2.0/eclipselink-2.2.0.v20110202-r8913.zip>, che decomprimeremo in una cartella chiamata EclipseLink.

Il release bundle dell'ultima versione di **Hibernate**, nel nostro caso la 3.6.2.Final, da <http://sourceforge.net/projects/hibernate/files/hibernate3/3.6.2.Final/hibernate-distribution-3.6.2.Final-dist.tar.gz/download>, da decomprimere in una cartella Eclipse.

L'ultima distribuzione di **SLF4J**³, qui la 1.6, da <http://www.slf4j.org/dist/slf4j-1.6.1.tar.gz> da decomprimere in slf4j.

Il driver **JDBC per PostgreSQL** nella versione 9.0 per JDBC 4 (verificare i requisiti relativi alla versione del server PostgreSQL e della JVM in uso) da <http://jdbc.postgresql.org/download/postgresql-9.0-801.jdbc4.jar>.

A questo punto iniziamo a creare le due librerie: nella sezione *JPA implementation*, al fianco della lista delle librerie utilizzabili, è presente il pulsante *Manage libraries...* che apre il pannello di configurazione delle librerie.

Procediamo creando una libreria, che chiameremo *EclipseLink*, che conterrà i seguenti files:

eclipselink.jar dalla cartella EclipseLink/jlib;

javax.persistence_2.0.3.v201010191057.jar da EclipseLink/jlib/jpa;

postgresql-9.0-801.jdbc4.jar che a rigor di logica non farebbe parte delle librerie relative all'implementazione JPA dell'EclipseLink, ma è comunque necessario al funzionamento del provider.

Creiamo una seconda libreria, chiamata *Hibernate*, che conterrà:

hibernate3.jar dalla cartella Hibernate;

hibernate-jpa-2.0-api-1.0.0.Final.jar da Hibernate/lib/jpa;

tutti i jar da Hibernate/lib/required;

slf4j-api-1.6.1.jar e slf4j-nop-1.6.1.jar da slf4j.

³Il **SIMPLE LOGGING FACADE FOR JAVA** è uno strumento di facade che media l'accesso a vari frameworks di logging[20], e viene appunto richiesto da Hibernate per supportare le funzionalità di logging

A questo punto confermiamo con *Ok*, una volta tornati al pannello precedente abilitiamo le due librerie appena create.

Dobbiamo ora configurare la sezione *Connection*: clicchiamo su *Add connection...*, nel pannello che si aprirà selezioniamo *PostgreSQL*, infine nel campo *Name* inseriamo *PostgreSQL*.

Il pulsante *Next>* ci porta sul pannello successivo dove selezioniamo *PostgreSQL JDBC Driver* (eventualmente modifichiamo le opzioni per fornire il path al driver JDBC che abbiamo precedentemente scaricato) e compiliamo i campi della connessione JDBC.

Clicchiamo su *Finish*, verifichiamo che sia selezionato il driver che abbiamo appena creato, e clicchiamo nuovamente su *Finish* per terminare le opzioni preliminari.

3.1.3 Creazione dell'entità

Iniziamo creando la classe che rappresenta l'entità Progetto (codice in appendice A.4), che andrà gestita dalla persistenza su database. Si tratta della trasposizione JPA di una semplice entità che contiene i due campi basilari *id* e *descrizione*, a cui aggiungiamo un semplice *toString* per la visualizzazione del flusso del programma.

Dovremmo ora creare una relazione sul database PostgreSQL che rispecchi la struttura dell'entità che abbiamo appena creato: Eclipse ci risparmia questa procedura puramente meccanica tramite un tool di auto-composizione. E' sufficiente cliccare con il tasto destro del mouse sul progetto in uso, selezionare *JPA Tools > Generate Tables from Entities...* e lo strumento si occuperà di generare le query di *CREATE TABLE*.

3.1.4 Implementazione del provider CRUD generico

A questo punto procediamo creando una interfaccia *CrudProvider* (vedi appendice A.2) che verrà successivamente implementata in *CrudProviderImpl* (vedi appendice A.3). L'interfaccia serve a fissare le idee sul comportamento che dovrà avere il provider CRUD che andremo a realizzare: contiene la definizione parametrica dei metodi **CREATE - RETRIEVE - UPDATE - DESTROY**.

La classe ne definisce una implementazione parametrica, che al momento utilizziamo unicamente per gestire la persistenza dell'entità Progetto, ma che potrà essere sperabilmente riutilizzata per ulteriori entità che verranno definite negli step successivi dei test. Tale implementazione mappa le istruzioni del modello CRUD nei metodi messi a disposizione da JPA, gestendo coerentemente gli *EntityManager* e le transazioni.

3.1.5 Uso di EclipseLink

Dopo aver definito gli obiettivi e lo scheletro logico dell'applicazione di test passiamo a testare l'effettivo funzionamento su un provider JPA: come primo step si è scelto di utilizzare EclipseLink, confidando in una maggiore semplicità di sviluppo rispetto ad Hibernate.

Come detto in precedenza, JPA richiede la configurazione di un *PersistenceProvider* per poter svolgere il proprio ruolo, che viene specificata all'interno del file `persistence.xml` (vedi appendice A.7).

Possiamo notare che vengono specificate tre diverse *persistence-unit*: focalizziamoci sulla prima, denominata appunto `ECLIPSE_LINK`, e osserviamo come basti specificare la classe che realizza il provider, la lista delle classi che devono essere gestite dalla persistenza e una lista di proprietà collegate con il data source. Ovviamente le altre due *persistence-unit* non sono necessarie al funzionamento del test a questo livello ma serviranno per gli step successivi.

Per verificare il lancio dei metodi e il loro effettivo funzionamento si è costruito un metodo *main* (vedi appendice A.1) molto semplice che sfrutta la classe `EntityGenerator` per costruire gli oggetti da persistere, richiamare i 4 metodi del modello CRUD e visualizzare in input il *toString* degli oggetti coinvolti.

Ci si è quindi basati sul confronto tra i messaggi visualizzati in output dal programma di test e i dati caricati sulle tabelle del database per verificare l'effettivo funzionamento del test.

Ecco il risultato di una esecuzione del programma

```
[EL Info]: 2011-03-25 14:01:38.745--ServerSession(783353674)--  
EclipseLink, version: Eclipse Persistence Services - 2.2.0.  
v20110202-r8913
```

```
[EL Info]: 2011-03-25 14:01:38.965--ServerSession(783353674)--  
file:/home/pier/EclipseProjects/test-crud-single/war/WEB-  
INF/classes/_ECLIPSE_LINK login successful
```

```
Inizio Test tabella singola: ECLIPSE_LINK
```

```
C Progetto [id=2, description=Nuova, hash=1644838203]
```

```
R Progetto [id=2, description=Nuova, hash=1644838203]
```

```
U Progetto [id=2, description=Modificata, hash=2055000954]
```

```
Fine Test
```

```
[EL Info]: 2011-03-25 14:01:39.138--ServerSession(783353674)--  
file:/home/pier/EclipseProjects/test-crud-single/war/WEB-  
INF/classes/_ECLIPSE_LINK logout successful
```

I record inseriti o modificati dal progetto di test è visibile in questo screenshot di pgAdmin⁴.

	id [PK] serial	descrizione character varying(255)
1	2	Modificata
*		

Figura 3.1: Dati inseriti da *test-crud-single* su provider EclipseLink

3.1.6 Uso di Hibernate

Una volta verificato il funzionamento dell'EclipseLink, si è voluto verificare se l'utilizzo di un ORM diverso avesse una qualche influenza nel codice JPA incluso nel progetto. In base alla lettura della documentazione, era lecito aspettarsi che fosse sufficiente la creazione di una seconda *persistence unit*, ad hoc per Hibernate, per far funzionare il tutto senza ulteriori modifiche.

Come si può vedere nel codice, per il semplice programma in esame il comportamento è stato esattamente quello atteso. E' bastato definire la seconda *persistence-unit* in *persistence.xml* (vedi appendice A.7), che riflette esattamente la struttura della prima a parte l'attributo *provider*.

Lo switch tra le due *persistence-unit* viene gestito tramite l'enumeratore *Provider* in *CrudProviderImpl.java* (vedi appendice A.2).

L'output dell'esecuzione del programma e gli effetti sulla persistenza sono identici a quelli della variante EclipseLink.

```
Inizio Test tabella singola: HIBERNATE

C Progetto [id=3, description=Nuova, hash=239228739]

R Progetto [id=3, description=Nuova, hash=239228739]

U Progetto [id=3, description=Modificata, hash=1009920161]

Fine Test
```

3.1.7 Migrazione su App Engine

Dopo le verifiche preliminari sulla comprensione dell'effettivo funzionamento di JPA, possiamo alla parte qualificante di questo test: la stima

⁴pgAdmin è una semplice GUI per l'amministrazione di un database PostgreSQL: rappresenta una utile interfaccia grafica per varie operazioni di gestione o interrogazione, come la modifica allo schema SQL, la creazione di query e la visualizzazione del contenuto delle tabelle.

	Id [PK] serial	descrizione character varying(255)
1	2	Modificata
2	3	Modificata

Figura 3.2: Dati inseriti da *test-crud-single* su provider Hibernate

del grado di difficoltà nella migrazione di un progetto preesistente verso l'architettura App Engine.

Il primo passo è la creazione di una directory *war* all'interno del progetto: possiamo ora includere le librerie relative ad App Engine, per questo entriamo nel pannello delle *Properties* relative al nostro *test-crud-single*, e andiamo modificare la parte relativa al sotto-menu *Google*.

Entriamo nella voce *App Engine*, dove abiliteremo il check-box relativo a *Use Google App Engine* e setteremo i parametri *Application ID* e *Version* coerentemente con l'*application id* che abbiamo ottenuto nel nostro profilo web.

Nella successiva voce *Web Application* verifichiamo che il check-box *This project has a WAR directory* sia abilitato, e che il path della directory corrisponda alla directory appena creata.

Osserviamo che alla chiusura del pannello la directory *war* viene popolata con i file JAR della libreria App Engine, e che Eclipse segnala due errori all'interno del progetto: la mancanza dei file *appengine-web.xml* e *web.xml*.

Eclipse prevede una *Quick fix* che genera automaticamente un template funzionante dei due file, possiamo quindi sfruttare questo utile strumento andando subito a modificare i due sorgenti creati come da appendice A.8 e A.9.

A questo punto il progetto è pronto per utilizzare App Engine, ma per l'accesso sfrutta ancora uno dei due ORM appena testati: se però proviamo a eseguirlo, si verifica subito il primo inconveniente: l'esecuzione fallisce in quanto si genera un'eccezione, nella quale riconosciamo il messaggio *ERROR: column jdodetachedstate of relation progettis does not exist*.

Ad un primo approccio, sembra che l'introduzione di App Engine abbia modificato la definizione dell'entità Progetto, aggiungendo un attributo *jdodetachedstate* che disturba i nostri ORM, in quanto non sono in grado di applicare la corretta mappatura delle colonne nella relazione sul database. Se verifichiamo il codice sorgente dell'entità non notiamo nessuna modifica, è quindi lecito aspettarsi che tale inconveniente sia dovuto a un qualche tipo di postprocessing che è stato introdotto in fase di compilazione.

Come abbiamo visto in precedenza l'accesso alla persistenza nel Datastore viene gestito attraverso il provider *DataNucleus*: tale provider si pone come un layer intermedio che traduce la sintassi di persistenza, JPA o JDO, nelle routine di accesso alle *Big Table*. Per far questo, inserisce appunto

una fase di postprocessing, tramite uno strumento denominato *DataNucleus Enhancer*, che crea una serie di metadati necessari alla traduzione in run-time.

All'interno di questa procedura, viene eseguito anche il *JDO Enhancement* [21] che, operando appunto a livello di bytecode, implementa le interfacce *PersistenceCapable* e opzionalmente *Detachable*: è proprio quest'ultima interfaccia a introdurre l'attributo `Object[] jdoDetachedState` che disturba gli ORM finora testati.

Per risolvere questo problema vi sono tre strade percorribili:

Abilitare o disabilitare l'Enhancer in base al target di esecuzione del progetto: l'Enhancer disabilitato consente l'utilizzo dei primi due ORM testati, ma genera una eccezione se si tenta di utilizzare App Engine correlata con la mancanza dei metadati, il comportamento è speculare abilitando il post-processing.

Modificare la relazione PostgreSQL aggiungendo allo schema una nuova colonna di tipo *bytea*, denominata appunto *jdodetachedstate*. Questo evita la generazione dell'eccezione, anche se inserisce dati non strettamente necessari nel database: nei successivi test si adotterà questa soluzione. Si noti come la ri-applicazione dello strumento di Eclipse *JPA Tools > Generate Tables from Entities...* accoppiata all'abilitazione del *DataNucleus Enhancer* inserisce automaticamente la nuova colonna nello schema delle tabelle generate.

Modificare l'entità Progetto aggiungendo le due righe

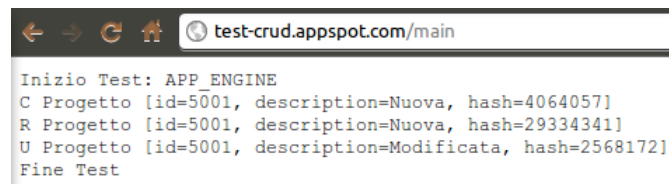
```
@Transient
public Object[] jdoDetachedState;
```

in modo da rendere *transient*, ossia non soggetto a persistenza, il campo *jdoDetachedState*: occorre però verificare se tale modifica va ad intaccare la logica di gestione delle entità in *DataNucleus*. Si riserva a una successiva analisi la praticabilità di tale opzione.

Dopo queste modifiche, il progetto è abilitato all'utilizzo di App Engine e non presenta malfunzionamenti con gli ORM precedentemente testati.

A questo punto è possibile completare il test, aggiungendo una ulteriore *persistence-unit* in *persistence.xml* (vedi appendice A.7), popolata con i parametri di accesso al Datastore, e definire una semplice servlet (vedi appendice A.5), in modo da reindirizzare i messaggi della classe di test dallo standard output allo stream di uscita visualizzabile nel browser.

In definitiva è possibile testare il funzionamento delle scritture sul Datastore sia attraverso l'esecuzione locale tramite il plugin App Engine per Eclipse, sia sul cloud effettuando il deployment nel dominio appspot.



```
Inizio Test: APP_ENGINE
C Progetto [id=5001, description=Nuova, hash=4064057]
R Progetto [id=5001, description=Nuova, hash=29334341]
U Progetto [id=5001, description=Modificata, hash=2568172]
Fine Test
```

Figura 3.3: Messaggi di output di *test-crud-single* su provider App Engine

Il risultato dell'esecuzione sul cloud non differisce dalle varianti su database relazionale.

Verifichiamo l'effettivo funzionamento dei salvataggi su persistenza attraverso il *Datastore viewer* di App Engine.



ID/Name	description
id=2002	Modificata
id=4001	Modificata
id=5001	Modificata

Delete

Figura 3.4: Dati inseriti da *test-crud-single* su provider App Engine

3.1.8 Osservazioni

Relativamente alla gestione di una semplice entità, osserviamo che JPA svolge egregiamente il suo compito di schermare i dettagli implementativi dell'accesso a persistenza.

Nel passaggio tra EclipseLink e Hibernate è stato necessario unicamente definire una nuova *persistence-unit* con i parametri del nuovo provider.

Nell'introduzione dell'accesso al datastore di AppEngine, l'unica difficoltà di migrazione è quella dovuta all'enhancement dell'entità che introduce un nuovo attributo, e che può essere risolta con relativa semplicità. Fatta salva questa problematica, tutto il lavoro che è stato richiesto nella migrazione è correlato alla configurazione di App Engine: la creazione della *persistence-unit* specifica e delle classi necessarie al funzionamento della servlet sul cloud.

3.2 Peristenza di una relazione uno-a-molti

3.2.1 Descrizione

Lo scopo del test è di estendere le verifiche sulla riutilizzabilità e compatibilità già impostate nel progetto precedente, nel caso in cui non si abbia a che fare con una entità isolata ma dove esista una relazione di tipo uno-a-molti tra due entità.

Anche in questo caso si mantiene l'approccio CRUD, opportunamente adattato in modo da evidenziare la gestione del collegamento tra le due entità.

3.2.2 Modifiche alle entità

Nella test precedente, si aveva a che fare con una unica entità Progetto, formata solo dai campi *id* e *descrizione*: in questo step si aggiunge una ulteriore entità, denominata Attività, anch'essa banalmente formata dai campi *id* e *descrizione*.

La parte significativa del test è la presenza di una relazione uno-a-molti tra le due entità, che viene gestita attraverso un attributo *progetto* nella classe Attività, contrassegnato dall'annotazione *@ManyToOne* (vedi appendice A.12).

Tale attributo comporta la presenza di una lista di Attività all'interno della classe Progetto, in questo caso contrassegnata dall'annotazione *@OneToMany* (*cascade=CascadeType.ALL*, *mappedBy = progetto*, *orphanRemoval = true*) (vedi appendice A.11). Risulta molto utile osservare come tale lista sia mappata dall'attributo *progetto* dall'altro lato della relazione, in questo modo la lista viene automaticamente gestita e popolata da JPA, ciò comporta anche la propagazione delle modifiche in *cascade* e l'eliminazione delle attività collegate quando si elimina un Progetto, garantendo così il vincolo di chiave esterna.

Come conseguenza dell'aggiunta di una nuova entità, è stato necessario modificare le tre *persistence-unit* definite in *persistence.xml*, per specificare che anche la classe Attività deve essere gestita dalla persistenza.

3.2.3 Modifiche al codice di testing

Le modifiche più sostanziali al codice sono quelle relative alla classe *EntityGenerator* (vedi appendice A.13), che genera le entità da salvare in persistenza. Tali modifiche riguardano la generazione di una nuova Attività durante il codice di *Create*, che viene correlata al progetto proprio grazie alla relazione *@OneToMany*.

Si verifica successivamente come questa Attività venga automaticamente ricaricata durante la fase di *Retrieve*, che sia coerentemente aggiornata dall'*Update* e che venga eliminata in fase di *Destroy* del Progetto.

Le modifiche introdotte al codice nei due punti appena presentati sono correlate con il cambio di logica del programma di test, dove si passa alla gestione di due entità collegate da una relazione di tipo uno-a-molti. Si tratta dunque di modifiche *necessarie* in relazione al cambio di obiettivi del codice di test, ma già *sufficienti* per un corretto funzionamento: con questa configurazione il progetto è già operativo, tanto che se ne può testare il funzionamento con EclipseLink e Hibernate.

Ecco ad esempio il risultato di due esecuzioni

```
[EL Info]: 2011-04-07 09:20:33.221--ServerSession(1863318328)--
EclipseLink, version: Eclipse Persistence Services - 2.2.0.
v20110202-r8913
```

```
[EL Info]: 2011-04-07 09:20:33.426--ServerSession(1863318328)--
file:/home/pier/EclipseProjects/test-crud-one-to-many/war/
WEB-INF/classes/_ECLIPSE_LINK login successful
```

Inizio Test one to many: ECLIPSE_LINK

C Progetto [id=1, description=Nuovo, hash=176065613]

Attivita [id=1, description=Nuova, progetto=1, hash
=779185335]

R Progetto [id=1, description=Nuovo, hash=176065613]

Attivita [id=1, description=Nuova, progetto=1, hash
=779185335]

U Progetto [id=1, description=Modificato, hash=176065613]

Attivita [id=1, description=Nuova, progetto=1, hash
=779185335]

D Progetto [id=1, description=Modificato, hash=176065613]

Attivita [id=1, description=Nuova, progetto=1, hash
=779185335]

Fine Test

```
[EL Info]: 2011-04-07 09:20:33.646--ServerSession(1863318328)--
file:/home/pier/EclipseProjects/test-crud-one-to-many/war/
WEB-INF/classes/_ECLIPSE_LINK logout successful
```

Inizio Test one to many: HIBERNATE

C Progetto [id=2, description=Nuovo, hash=264829771]

\quad Attivita [id=2, description=Nuova, progetto=2, hash
=1433965066]

```

R Progetto [id=2, description=Nuovo, hash=264829771]

\quad Attivita [id=2, description=Nuova, progetto=2, hash
=1433965066]

U Progetto [id=2, description=Modificato, hash=264829771]

\quad Attivita [id=2, description=Nuova, progetto=2, hash
=1433965066]

D Progetto [id=2, description=Modificato, hash=264829771]

\quad Attivita [id=2, description=Nuova, progetto=2, hash
=1433965066]

Fine Test

```

	id [PK] serial	descrizione character varying(255)	jdodetachedstate bytea
1	1	Modificato	<dati binari>
2	2	Modificato	<dati binari>
*			

Figura 3.5: Dati inseriti nella tabella Progetto

	id [PK] integer	descrizione character varying(255)	jdodetachedstate bytea	progetto_id bigint
1	1	Nuova	<dati binari>	1
2	2	Nuova	<dati binari>	2
*				

Figura 3.6: Dati inseriti nella tabella Attivita

3.2.4 Problema: la gestione delle chiavi di AppEngine

Dopo la verifica del corretto funzionamento del nostro progetto sui due ORM locali, possiamo ai test sulla trasposizione in App Engine. Abilitandone le librerie, il tentativo di eseguire il codice viene a generarci una eccezione nella quale riconosciamo il messaggio

```
Error in meta-data for entities.Attività.id: Cannot have a java.lang.Long primary key and be a child object (owning field is entities.Progetto.attività).
```

La comparsa di un problema di compatibilità non ci stupisce: infatti nella sezione 2.3, relativa al Datastore, abbiamo messo in luce come l'utilizzo delle relazioni uno-a-molti comporti una serie di vincoli nella gestione delle entità su persistenza.

Si rimanda all'apposita sezione per approfondimenti: in questo contesto si evidenzia come l'eccezione sollevata sia dovuta al fatto che una *child entity*, che nel nostro caso è l'Attività, nel datastore preveda una *key* che le consenta di mantenere un riferimento alla *parent entity*, ossia al progetto al quale è collegata.

Dopo una serie di prove, si è trovata una soluzione soddisfacente ridefinendo la parte di codice che specificava il comportamento della *key* dell'entità Attività, passando da una struttura del tipo

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
```

ad una più complicata

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Extension(vendorName=datanucleus, key=gae.encoded-pk,
value=true) //richiesto dalla gestione delle key in AppEngine
@Convert(longToStringEL)
@Type(type=longToStringHI)
private String id;
```

Analizziamo brevemente le modifiche introdotte:

- è stato modificato il tipo dell'attributo *id*, che passa da *Long* a *String*: si noti come tale variazione debba coinvolgere solo il modo in cui l'attributo viene gestito in memoria dal codice, e non debba idealmente tradursi in una modifica dello schema del database relazionale utilizzato dagli ORM locali;
- è stata introdotta una annotazione *@Extension*: tale annotazione ha effetto unicamente sul comportamento della *persistence-unit* che va ad operare su App Engine, e permette a *DataNucleus* di accoppiare le chiavi generate dal datastore all'attributo ID, che vengono quindi gestite dal codice come una semplice stringa;
- a questo punto il progetto è funzionante in App Engine, ma si perde la compatibilità con gli ORM locali, in quanto si viene a perdere il mapping diretto tra l'attributo ID di tipo *String* dell'entità e il campo ID di tipo numerico della tabella del database relazionale: dato che sarebbe preferibile non modificare lo schema del database, si è optato per ulteriori annotazioni nel codice;

3.2.5 Osservazioni

Possiamo osservare come JPA gestisca in modo completo e intuitivo le relazioni di tipo uno-a-molti: come punto a favore si può includere la ridotta quantità di codice necessaria ad ampliare il progetto per gestire la relazione tra le entità, evidente soprattutto dal fatto che non è stato necessario modificare la classe `CrudProviderImpl` (vedi appendice A.3) che gestisce la persistenza.

Di conseguenza l'ampliamento del progetto fino ad arrivare al test sui due ORM locali è stata decisamente lineare. Per quanto riguarda il passaggio su App Engine, la necessità di ridefinire il tipo della chiave nella *child entity* era ampiamente previsto in base alla documentazione ed è stato velocemente risolto aggiungendo l'annotazione richiesta.

A questo punto si sono verificati i problemi nella compatibilità con EclipseLink e Hibernate, che hanno richiesto una certa fase di studio in quanto i due strumenti richiedono un diverso set di annotazioni specifiche, poiché il mapping tra tipi di dato diversi non viene nativamente gestito da JPA ma viene delegato a estensioni di ciascuno strumento.

La soluzione presentata sembra comunque soddisfacente, in quanto si riesce a mantenere la compatibilità dello stesso codice sui tre diversi *persistence provider* e la parte di codice da modificare è limitata alla definizione della classe relativa alla *child entity*.

Come ultima nota si riporta che è stato necessario includere una parte delle librerie di Hibernate anche nella *directory war* che viene poi caricata su App Engine, in quanto la classe `Attivita` contiene un riferimento alla classe `HibernateConverter` (vedi appendice A.15), che a causa dell'utilizzo delle classi della libreria Hibernate genera una eccezione nel *classloader* se tali file non vengono inclusi.

3.3 Peristenza di una relazione multi-a-molti

3.3.1 Descrizione

Questo progetto chiude idealmente la parte dei test CRUD su JPA: si introduce quindi la sintassi necessaria a supportare le relazioni multi-a-molti tra le entità, e si vanno a verificare i problemi che si vengono a creare nel trasferimento su App Engine.

E' possibile escludere a priori un funzionamento completo del test: dalla documentazione sappiamo che il Datastore di App Engine non supporta questo tipo di relazioni. Limitiamo al momento l'analisi alla sola modifica delle entità, e all'implementazione del solo codice di *Create* con successivo test su ORM locali e su App Engine.

3.3.2 Modifica alle entità e all'Entity Generator

Rispetto al progetto precedente, viene eliminata l'entità *Attivita*, mentre si mantiene il *Progetto* (vedi appendice A.16), al quale viene affiancata una entità *Addetto* (in appendice A.17).

Queste due entità sono legate da una relazione multi-a-molti: all'interno del *Progetto* troveremo quindi una lista degli addetti impegnati, viceversa all'interno dell'*Addetto* troveremo una lista dei progetti su cui sta lavorando.

Entrambi i lati della relazione sono gestiti dall'annotazione *@ManyToMany*, che dal lato del progetto mantiene gli attributi relativi al *CascadeType* e all'entità proprietaria della relazione, che in questo caso assegnamo all'addetto.

Le modifiche all'Entity Generator (vedi appendice A.18) e al file *persistence.xml* (in appendice A.19) sono minimali, e riguardano un leggero refactoring in quanto eliminiamo la gestione dell'entità *Attivita* e introduciamo l'*Addetto*.

Al momento disabilitiamo i metodo *retrieve*, *update* e *destroy* in quanto ci concentriamo sul semplice test della funzionalità sull'ORM locale, e sugli errori che si verificano nel tentativo di porting su App Engine

3.3.3 Risultati

Attraverso le semplici modifiche appena illustrate, è immediato ottenere il seguente risultato dall'esecuzione su un ORM locale, in questo caso Hibernate:

```
Inizio Test many to many: HIBERNATE
C Progetto [id=1, description=Nuovo progetto, hash=937777044]
  Addetto [id=1, description=Nuovo addetto, progetti=1, hash
    =280564915]
Fine Test
```

Riportiamo il risultato dell'inserimento nelle tabelle del database PostgreSQL, ricordando che la gestione di una associazione multi-a-molti nel modello relazionale richiede la creazione di una ulteriore tabella, che contiene appunto le chiavi esterne necessarie alla memorizzazione degli accoppiamenti.

	id [PK] serial	descrizione character varying(255)	jdodetachedstate bytea
1	1	Nuovo addetto	<dati binari>
*			

Figura 3.10: Dati inseriti nella tabella Addetto

	id [PK] serial	descrizione character varying(255)	jdodetachedstate bytea
1	1	Nuovo progetto	<dati binari>
*			

Figura 3.11: Dati inseriti nella tabella Progetto

	progetti_id [PK] bigint	addetti_id [PK] bigint
1	1	1
*		

Figura 3.12: Dati relativi alla relazione multi-a-molti

Nel momento in cui si tenta di eseguire il codice in esame su App Engine, si ottiene il lancio di una eccezione nella quale riconosciamo il seguente messaggio di errore: *Error in meta-data for entities.Progetto.addetti: Many-to-many is not currently supported in App Engine.*

Si tratta di un messaggio ampiamente atteso, che ci ricorda che la relazione multi-a-molti da noi introdotta non sia supportata dal Datastore.

3.3.4 Osservazioni

All'interno dell'eccezione sollevata dal Datastore, oltre al messaggio precedentemente riportato, compare anche il suggerimento *As a workaround, consider maintaining a List<Key> on both sides of the relationship.* See http://code.google.com/appengine/docs/java/datastore/relationships.html#Unowned_Relationships for more information.

Come abbiamo visto nella documentazione, uno delle possibili alternative all'utilizzo delle annotazioni `@ManyToMany` è rappresentato dalla

possibilità di memorizzare da ciascun capo della relazione una lista delle chiavi delle entità associate.

Questa soluzione è percorribile anche nel caso in esame, ma non verrà approfondita in quanto richiede la costruzione di una parte di codice non banale per gestire manualmente il comportamento della relazione.

Oltre alla perdita degli automatismi, vi sono una serie di problemi che devono essere considerati dal codice aggiunto, in quanto App Engine non supporta la modifica di entità che non facciano parte dello stesso *entity group* all'interno di una transazione.

Come ultimo appunto, si osserva che l'aggiunta delle due liste di chiavi non soddisfa il requisito della portabilità inversa, ossia da App Engine verso gli ORM locali, in quanto la classe Key fa parte delle librerie di App Engine.

In definitiva, l'utilizzo delle relazioni multi-a-molti comporta grosse difficoltà nel porting verso App Engine, oppure verso gli ORM locali, in quanto richiede la riscrittura di parti non banali di codice.

4 UTILIZZO DI JDBC SU APP ENGINE

I test funzionali del capitolo precedente hanno messo in luce l'esistenza di limitazioni e vincoli, che uno sviluppatore si trova ad affrontare nel caso in cui realizzi la propria logica di persistenza secondo uno dei paradigmi più utilizzati, il modello relazionale.

Verranno ora proposte due alternative, con lo scopo di aumentare le funzionalità di persistenza utilizzabili su App Engine, analizzando come la loro introduzione può consentire l'utilizzo di pratiche consolidate nella persistenza dei dati.

4.1 Emulazione di JDBC tramite JPQL

4.1.1 Descrizione

Lo scopo del progetto è verificare il grado di compatibilità esistente tra la sintassi indotta da JDBC¹ e da JPA nell'accesso ai dati in persistenza, con riferimento sia ai metodi che vengono invocati sugli oggetti della libreria *java.sql*, sia alle sintassi delle query SQL che vengono inviate al database.

L'obiettivo ideale sarebbe costruire una insieme di classi di *bridging*, con firma identica a quelle utilizzate da JDBC, che si occupino di tradurre i metodi richiesti in chiamate alle routine di JPA: se questo fosse possibile, si sarebbe idealmente in grado di utilizzare un codice che utilizza la logica JDBC sul datastore di AppEngine, semplicemente modificando le sezioni di *import* del codice Java e utilizzando le classi di *bridging*.

All'interno del progetto si andrà quindi a compiere un rapido studio di fattibilità, implementando le funzioni minimali del *bridge* per supportare un semplice programma di prova, che invierà una serie di query SQL ad entrambi gli strumenti per verificare i risultati ottenuti.

Dato che App Engine introduce delle limitazioni al formato delle query ammissibili, che esula dallo standard, il test deve essere replicato su 3 scenari differenti, nei quali il codice basato sulla sintassi JDBC viene testato su:

- driver JDBC su database locale PostgreSQL;
- *bridge* su JPA (*persistence-unit* su database locale PostgreSQL);
- *bridge* su JPA (*persistence-unit* sul DataNucleus di App Engine).

¹JAVA DATABASE CONNECTIVITY è una API Java per l'accesso ai database, con particolare riferimento a quelli relazionali.

4.1.2 Nozioni preliminari

La logica di accesso al database indotta da JDBC presenta alcune differenze con l'accesso tramite ORM e JPA che abbiamo presentato nei progetti precedenti. In un codice basilare basato su questa sequenza, si osserva generalmente la seguente struttura:

- è necessario disporre di un driver JDBC che consenta l'accesso al particolare DBMS che si intende utilizzare;
- attraverso l'oggetto *DriverManager* si stabilisce una connessione al database, che viene gestita attraverso l'oggetto *Connection*;
- su una connessione esistente si crea un oggetto *Statement*, grazie al quale si va successivamente a eseguire una query SQL;
- se la query rappresenta una interrogazione (*SELECT*) del database, viene ritornato un oggetto di tipo *ResultSet* che veicola i record rispondenti ai criteri di interrogazione.

Nei progetti precedenti, basati su JPA, le ricerche su database sfruttavano il metodo *find* dell'oggetto *EntityManager*, mentre le varie modifiche alla persistenza si basavano sui vari *persist*, *merge* e *remove*. L'approccio basato su questi metodi risulta difficilmente applicabile alle classi che vorremmo costruire, in quanto richiederebbe lo sviluppo di un'architettura di traduzione dalle query testuali ai metodi di persistenza con una complessità che esula dagli scopi di questo test.

Si è quindi scelto di basarsi su un'altra componente di JPA, ossia del linguaggio JPQL²: si tratta del linguaggio definito per l'accesso alle entità in persistenza, descritto in dettaglio in [22].

Al pari di SQL, JPQL consente di specificare in modo testuale sia i criteri di ricerca per le interrogazioni, sia le modifiche da operare sui dati.

Sia però chiaro che **JPQL non è SQL**: SQL opera infatti sugli oggetti di un database relazionale, mentre JPQL opera sulle entità definite in JPA. Nonostante la sintassi sia estremamente simile, tanto da permettere in molti casi l'applicazione diretta della stessa query ai due contesti, esistono dei casi in cui tale sintassi diverge e altri in cui le funzionalità previste in un caso non sono disponibili in un altro.

Per una trattazione delle differenze di sintassi, si rimanda alla sottosezione 4.1.4 dove vengono presentati e commentati i risultati dell'applicazione di un set di query nei vari contesti.

²JAVA PERSISTENCE QUERY LANGUAGE

4.1.3 Le classi di *bridging*

Presentiamo ora il funzionamento ad alto livello delle classi di *bridging* che sono state costruite all'interno del progetto: tali specifiche non devono ritenersi esaustive in quanto l'implementazione proposta è minimale e relativa al supporto al semplice codice di prova.

DriverManager (vedi appendice A.22): nel nostro caso, l'unico metodo che viene sfruttato in questa classe è il *getConnection*, che solitamente si occuperebbe di caricare il driver JDBC e i parametri di connessione. Dato che in JPA la connessione non viene esplicitamente stabilita ma si passa per l'*EntityManager*, il metodo da noi realizzato non fa altro che invocare il costruttore della classe *Connection* da noi definita e restituire il riferimento al nuovo oggetto creato.

Connection (vedi appendice A.21): la nostra connessione non sfrutta i parametri della connessione JDBC, ma si basa unicamente sull'oggetto *EntityManager*, che viene inizializzato come attributo dell'oggetto dal costruttore.

Per il test si sfrutta l'unico metodo *createStatement*, che invoca appunto il costruttore di *Statement*; è stato aggiunto un attributo *provider*, comprensivo dei metodi *getter* e *setter* che consente di variare il *persistence provider* al fine di diversificare lo scenario basato su ORM locale (in questo caso Hibernate) da quello App Engine.

Incidentalmente, si può notare come il *persistence provider* su ORM locale faccia comunque uso di alcuni parametri di connessione JDBC: all'interno di *persistence.xml* si specificano infatti i parametri di configurazione del provider basato su Hibernate, che sfrutta una connessione JDBC mascherata dalla logica JPA.

Statement (vedi appendice A.25): il nostro codice di prova richiede l'implementazione dei due metodi *executeQuery* e *execute*, che richiedono entrambi un parametro di tipo *String* e vanno appunto ad eseguire la query.

Il primo metodo, che noi utilizziamo per le query di interrogazione, ritorna inoltre un oggetto *ResultSet* che veicola i record del risultato, mentre il secondo viene da noi utilizzato unicamente per le query di aggiornamento.

Si ricorda che, se nello *Statement* della libreria *java.sql* il parametro *query* permette il passaggio di una query SQL che verrà eseguita sul DBMS, in questa classe di *bridging* assume il significato di una query JPQL, che va ad agire sulle entità di persistenza sfruttando l'*EntityManager* che proviene dalla connessione.

ResultSet (vedi appendice A.23): nella libreria *java.sql*, questo oggetto è realizzato per veicolare i risultati di una interrogazione SQL, ed eventualmente per modificarli, ma nel nostro caso vengono utilizzati solo due metodi:

next che avanza il cursore all'interno della lista dei record;

getString che accetta un parametro *index*, e estrae un oggetto di tipo *String* in corrispondenza dell'attributo nella posizione *index* sul record puntato dal cursore.

Una query JPQL ritorna invece una serie di risultati sotto forma di un unico *Object*, oppure di un array di *Object*, si è pertanto predisposto il codice necessario ad accoppiare le logiche attraverso una serie di *cast* e di navigazioni su liste.

SQLException (vedi appendice A.24): si tratta di una semplice eccezione che veicola gli errori che possono verificarsi durante l'esecuzione del *bridge*, in questo modo i blocchi *try-catch* del codice originale possono essere mantenuti inalterati.

4.1.4 Analisi delle query

I test sulle query nei tre scenari vengono svolti attraverso l'oggetto *QueryTester* (vedi appendice A.27), che viene sfruttato dalla classe *Main* (in appendice A.20) per mettere a confronto i risultati ottenuti dal driver JDBC rispetto a quelli dovuti al *bridge* che opera sfruttando *Hibernate*, e dalla *MainServlet* (in appendice A.26) che viene eseguita su App Engine per testare il *bridge* sul *Datastore*.

Il *QueryTester* inizializza una array di query che si ritengono significative per il progetto in esame, e poi va ad eseguirle sui diversi scenari, occupandosi di visualizzare i valori di ritorno o i messaggi di errore.

Si presentano inoltre i messaggi completi ottenuti dalle esecuzioni nei diversi scenari, che vengono raggruppati al fine di migliorare la leggibilità (i risultati provengono da due sorgenti diverse in quanto per gli scenari locali si sfrutta un'applicazione Java, mentre per l'App Engine si va ad eseguire la servlet sulla piattaforma di cloud):

```
1) SELECT * FROM Progetto p
   SUCCESS
   1
   2
   FAIL: org.hibernate.hql.ast.QuerySyntaxException: unexpected
        token: * near line 1, column 8 [SELECT * FROM entities.
        Progetto p]
   FAIL: Identifier expected at character 1 in "*"
-----
```

```
2) SELECT p FROM Progetto p
SUCCESS
  (1,"Progetto di prova",)
  (2,"Secondo progetto",)
SUCCESS
  Progetto [id=1, description=Progetto di prova, hash
            =1830423861]
  Attività [id=1, description=Nuova, progetto=1, hash
            =107865609]
  Progetto [id=2, description=Secondo progetto, hash
            =694666723]
SUCCESS
  Progetto [id=32, description=Modificato, hash=544581444]
  Attività [id=
            ag10ZXN0LWNydWRyHAsSCFByb2dldHRvGCMCxEEQXR0aXZpdGEYIw
            , description=Nuova, progetto=32, hash=1795574656]
  Progetto [id=34, description=Modificato, hash=1783035748]
  Attività [id=
            ag10ZXN0LWNydWRyHAsSCFByb2dldHRvGCMCxEEQXR0aXZpdGEYIw
            , description=Nuova, progetto=34, hash=1495006398]
-----

3) SELECT descrizione FROM Progetto
SUCCESS
  Progetto di prova
  Secondo progetto
SUCCESS
  Progetto di prova
  Secondo progetto
SUCCESS
  Modificato
  Modificato
-----

4) SELECT p.descrizione FROM Progetto p
SUCCESS
  Progetto di prova
  Secondo progetto
SUCCESS
  Progetto di prova
  Secondo progetto
SUCCESS
  Modificato
  Modificato
-----

5) SELECT a.descrizione FROM Progetto p INNER JOIN Attività a
   ON p.id=a.progetto_id
SUCCESS
  Nuova
FAIL: org.hibernate.hql.ast.QuerySyntaxException: unexpected
      token: ON near line 1, column 69 [SELECT a.descrizione
      FROM entities.Progetto p INNER JOIN Attività a ON p.id=a.
      progetto_id]
```

```
FAIL: FROM clause has identifier Attivita but this is unknown
```

```
6) SELECT a.id FROM Progetto p INNER JOIN p.attivita a
FAIL: ERROR: syntax error at end of input
Posizione: 52
SUCCESS
1
FAIL: No meta-data for member named a.id on class entities.
Progetto. Are you sure you provided the correct member
name in your query?
```

```
7) SELECT p.id FROM Progetto p INNER JOIN p.attivita a
FAIL: ERROR: syntax error at end of input
Posizione: 52
SUCCESS
1
SUCCESS
32
34
```

```
8) SELECT MAX(p.id) FROM Progetto p
SUCCESS
2
SUCCESS
2
FAIL: Problem with query <SELECT MAX(p.id) FROM Progetto p>:
App Engine datastore does not support operator MAX.
```

```
9) SELECT p.id FROM Progetto p ORDER BY id DESC
SUCCESS
2
1
SUCCESS
2
1
SUCCESS
34
32
```

```
10) INSERT INTO Progetto VALUES (300,'Temp',NULL)
SUCCESS
FAIL: org.hibernate.hql.ast.QuerySyntaxException: expecting
OPEN, found 'VALUES' near line 1, column 22 [INSERT INTO
Progetto VALUES (300,'Temp',NULL)]
FAIL: JPQL Query should always start with SELECT/UPDATE/
DELETE
```

```
11) UPDATE Progetto p SET descrizione='updated' WHERE id=300
    SUCCESS
    SUCCESS
    SUCCESS
-----
12) DELETE FROM Progetto WHERE id=300
    SUCCESS
    SUCCESS
    SUCCESS
```

In tabella 4.1 vengono schematizzati i risultati dell'applicazione dei test nei tre contesti di interesse.

4.1.5 Osservazioni

Riportiamo una serie di conclusioni, che in parte rispecchiano le note riportate nella tabella di sintesi:

- la sintassi JPQL è molto simile a SQL, per query basilari non ci sono problemi di compatibilità, a parte l'impossibilità di utilizzare la *SELECT ** sotto JPA e la differenza nel tipo del valore di ritorno per l'identificatore di tabella/entità
- le tabelle del database devono avere lo stesso nome delle entità, perché la stessa stringa viene utilizzata sia nella sintassi SQL che richiede identificatori di tabella, sia in JPQL che richiede identificatori di entità;
- la presenza di JOIN comporta grossi problemi di compatibilità, proprio perché la sintassi indotta da tale clausola è differente;
- query contenenti JOIN che rispettano la sintassi JPQL non danno comunque garanzia di funzionare su App Engine, a causa di ulteriori limitazioni che vengono imposte nell'accesso alle *child entities*;
- le clausole di aggregazione non sono supportate su App Engine;
- le clausole di ordinamento non creano problemi nei tre scenari.

In definitiva l'utilizzo delle classi di *bridging* rappresenta una soluzione percorribile solo per query molto semplici e con vincoli estremamente invasivi, a meno di complicare in modo molto significativo la struttura di tali classi al fine di sopperire ad una parte di tali limitazioni.

Si può pensare ad un pre-processing delle query per sopperire alle differenze di sintassi: ad esempio, intercettando la presenza di *SELECT ** sarebbe possibile sostituirla con una sintassi alternativa valida che comprenda la lista di tutti gli attributi da estrarre, mentre intercettando le clausole di *JOIN* si arriverebbe a tradurle in modo compatibile con JPQL.

Tabella 4.1: Risultati dell'applicazione delle query

Query	JDBC	Hib.	GAE
Basilari			
SELECT * FROM Progetto ^a p	✓		
SELECT p ^b FROM Progetto p	✓	✓	✓
SELECT descrizione FROM Progetto	✓	✓	✓
SELECT p.descrizione FROM Progetto	✓	✓	✓
Join^c			
SELECT a.descrizione FROM Progetto p INNER JOIN Attivita a ON p.id=a.progetto_id	✓		
SELECT a.id FROM Progetto p INNER JOIN p.attivita a		✓	^d
SELECT p.id FROM Progetto p INNER JOIN p.attivita a		✓	✓
Aggregazione			
SELECT MAX(p.id) FROM Progetto p	✓	✓	^e
Ordinamento			
SELECT p.id FROM Progetto p ORDER BY id DESC	✓	✓	✓
Modifica			
INSERT INTO Progetto VALUES (300,'Temp',NULL)	✓	^f	^f
UPDATE Progetto p SET descrizione='updated' WHERE id=300	✓	✓	✓
DELETE FROM Progetto WHERE id=300	✓	✓	✓

^a Si noti come *Progetto* ha la duplice valenza di identificatore di tabella in SQL e di identificatore di entità in JPQL.

^b I valori ritornati dalla stessa query differiscono in quanto in SQL viene generata una stringa che comprende tutti gli attributi della tabella, mentre in JPQL si ritorna un riferimento alle entità *Progetto* che consente un pieno accesso agli attributi.

^c Si noti come la sintassi delle query contenenti JOIN è incompatibile tra SQL e JPQL.

^d La query sarebbe compatibile con lo standard JPQL, ma App Engine non supporta l'accesso ai campi delle *child entities* nelle JOIN.

^e App Engine non supporta le funzioni di aggregazione (GROUP BY, HAVING, SUM, AVG, MAX, MIN).

^f La sintassi JPQL non prevede le query di INSERT.

Al momento lo sviluppo di un simile strato di traduzione delle query viene solamente ipotizzata, si rimanda lo sviluppo implementativo ad una successiva analisi.

A tutto ciò si aggiungono i limiti di imposti dal Datastore App Engine, che introduce appunto delle limitazioni nel tipo di query accettabili, in quanto si può assistere a un insieme non banale di query conformi alla sintassi JPQL che non sono però supportate.

Per completare il quadro si deve evidenziare una forte difficoltà nel debug: non esistono strumenti automatici che consentano un controllo preventivo in fase di compilazione sulla compatibilità delle query utilizzate, anche per l'abitudine diffusa di costruire a run-time le stringhe da utilizzare come query attraverso manipolazione di stringhe.

4.2 Architettura

4.2.1 Descrizione

Il progetto che si andrà a sviluppare ora prevede la costruzione di una libreria Java che, eseguita all'interno di App Engine, sia in grado di emulare nel modo più completo possibile le funzionalità dello standard JDBC, a partire dalla struttura stessa del codice, al fine di fornire un accesso ai dati su un DBMS relazionale, a cui è possibile accedere tramite un web server esterno ad App Engine.

Come esplicitato dalla figura, il progetto comprende due parti complementari e separate, chiamate rispettivamente *nembo-db* e *nembo-gae* che, condividendo anche una parte del codice, scambiano messaggi tramite lo standard JSON per gestire metodi e valori di ritorno che vengono generati da un codice che acceda a un database tramite JDBC.

Il codice di prova, che genera una serie di query SQL e ne visualizza i risultati, è stato raccolto in un ulteriore progetto, chiamato *jdbc-on-nembo*, che sfrutta appunto la libreria *nembo-gae*: nelle prossime sezioni verranno presentate le componenti del software sviluppato, cercando di metterne in luce le interazioni.

4.2.2 JSON

JAVASCRIPT OBJECT NOTATION è un semplice formato di scambio dati: è nato in ambiente Javascript per il trasferimento di dati tra client e server, ma per la sua semplicità e compattezza è stato successivamente adottato anche da altri linguaggi, tra cui Java.

Rispetto ad XML, che rappresenta l'opzione più famosa in quest'ambito, JSON può appunto vantare una maggiore compattezza, che si traduce in un minore flusso di dati da trasmettere, a parità di informazione: questo

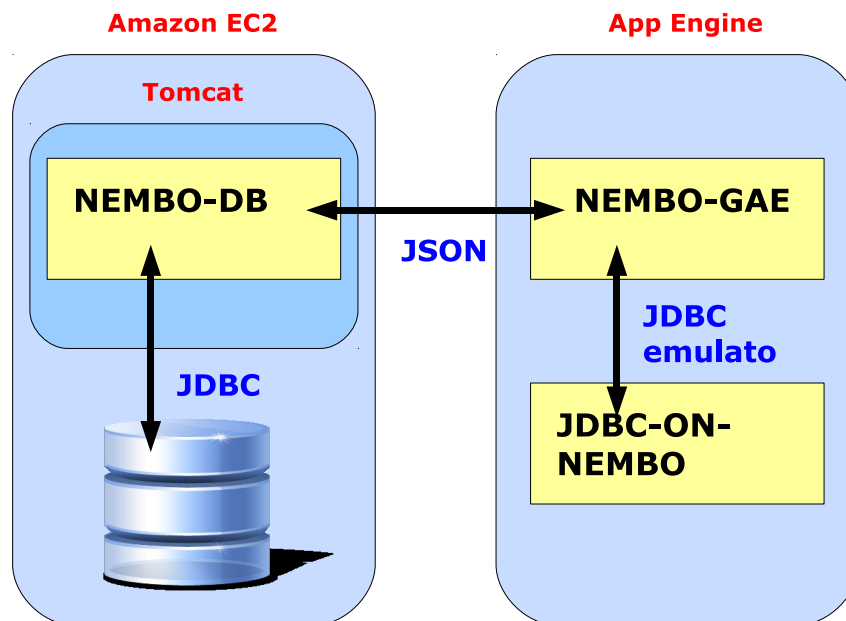


Figura 4.1: Schema di architettura di Nembo [7]

perchè vengono eliminati i marcatori di apertura e chiusura, a favore di una struttura interamente votata all'interscambio dei dati.

Nell'ambito web server, JSON può comunque vantare una buona diffusione, in merito delle sue caratteristiche che lo hanno reso a volte preferibile ad XML, tanto che alcuni fornitori di web service a volte predispongono l'accesso ai propri servizi secondo entrambe le modalità.

Una descrizione esaustiva dello standard si trova a [27]: è importante notare che non è però necessario conoscere interamente lo standard per sfruttare JSON all'interno di un proprio web service, in quanto esistono molteplici librerie che eseguono la traduzione e il parsing delle nostre strutture dati verso JSON.

Per il nostro progetto abbiamo utilizzato *google-gson*[28], una libreria che fornisce una implementazione dello standard JSON per il linguaggio Java, sviluppata dal team Google. Si tratta di un progetto relativamente recente (la prima release risale a metà 2008), che può vantare comunque dei frequenti aggiornamenti.

Nei test che sono stati effettuati *google-gson* ha dimostrato una buona semplicità d'uso e una buona stabilità, corredate da una documentazione puntuale, anche se in certi casi, che verranno discussi in seguito, ha richiesto la stesura di codice aggiuntivo rispetto alle previsioni per venire incontro a limitazioni progettuali.

4.2.3 *nembo-gae*

Il progetto *nembo-gae* si configura come una libreria destinata a contenere le classi qualificanti del progetto, ossia il codice che va ad emulare le corrispondenti classi delle API *java.sql*.

Tale libreria verrà successivamente compilata in un unico file *jar*, che dovrà essere distribuito sia sul server che ospita il DBMS relazionale (nel nostro caso dovrà dunque essere incluso nel progetto *nembo-db*), sia sul client App Engine che utilizzerà queste classi come un'emulazione dello standard JDBC.

Le classi sono raggruppate in un unico package, chiamato *com.gmail.conte.tesi*, che conterrà appunto una serie di classi che reimplementano le corrispondenti classi di *java.sql*, generando le chiamate ai web service quando si rende necessario l'accesso al database.

E' interessante notare come tali classi vadano direttamente ad implementare le corrispondenti interfacce dello standard: ad esempio *Nembo-Connection* va ad implementare l'interfaccia *java.sql.Connection*, allo scopo di definire il primo punto di accesso nella connessione JDBC emulata, mettendo a disposizione quindi gli stessi metodi, con identiche firme.

Analogamente, sono previste le implementazioni delle altre interfacce comunemente utilizzate in JDBC, come lo *Statement* che consente di andare ad eseguire una query, o il *ResultSet* che veicola i risultati dell'interrogazione (e nel nostro caso deve essere trasferito dal DBMS relazionale fino all'applicazione App Engine).

In questo modo la sintassi del codice che utilizza *nembo* subisce modifiche minime rispetto all'utilizzo di JDBC nativo, come vedremo in seguito, in quanto gran parte del codice che si scrive in un progetto JDBC è basato sulle interfacce di *java.sql*.

In certi casi non è stato possibile implementare determinati metodi previsti dalle interfacce: si è comunque cercato di evidenziare questi comportamenti predisponendo delle eccezioni esplicative che vengono sollevate quando si tenta di accedere a metodi non supportati.

4.2.4 *nembo-db*

Il progetto *nembo-db* rappresenta la servlet che realizza attivamente il web service di emulazione della libreria JDBC: tale servlet dovrà quindi appoggiarsi ad un web server (nei nostri test Apache), e avere accesso un DBMS relazionale (nel nostro caso PostgreSQL) al quale reindirizzare le chiamate JDBC.

Per sfruttare le classi di *nembo-gae*, è necessario importare il corrispondente file *jar* all'interno del progetto, in quanto gli oggetti che costituiscono i risultati delle chiamate vengono costruiti all'interno di questo progetto.

to, e successivamente serializzati tramite JSON. Allo stesso tempo è stato necessario importare la libreria di *google-gson*.

A questo punto si è costruita la servlet, definita come *TranslationServlet*, che costituisce l'interfaccia dei nostri web service.

Questa servlet riceve i parametri dei metodi JDBC tramite JSON, dopodichè costruisce gli effettivi accessi al DBMS relazionale: una volta ottenuti i risultati, provvede a serializzarli e ritornarli al client.

In caso di errori la stessa eccezione viene serializzata tramite JSON, in modo da fornire messaggi utili all'utente (ad esempio, nel caso in cui la query SQL non sia sintatticamente corretta).

La servlet mantiene i riferimenti degli oggetti attivi, all'interno di una sezione *static*, sfruttando delle strutture dati *HashMap*: in questo modo, ogni volta che *nembo-gae* richiede l'inizializzazione di un oggetto (come una *Connection* o uno *Statement*, tale oggetto viene effettivamente costruito all'interno della *TranslationServlet* e mantenuto in memoria fino alla esplicita richiesta di deallocazione.

In base a questo approccio, viene comunicato al client l'*id* che permette di identificare all'interno della struttura dati l'oggetto creato, ma non vi è un effettivo passaggio degli oggetti non necessari.

Viceversa, nel caso in cui l'oggetto creato debba essere trasferito su *emphnembo-gae* (il caso principale è appunto il *ResultSet*, tale oggetto viene trasformato nella sua implementazione *NemboRowSet*, che viene poi serializzata e trasferita su App Engine, dove può essere ricostruita e utilizzata dal codice.

4.2.5 jdbc-on-nembo

Questo progetto assume il ruolo di test, in quanto costruisce una serie di chiamante al JDBC, secondo la sintassi JDBC ma sfruttando le classi di *nembo*, che verranno quindi redirezionate sul web service *nembo-db*.

In questo caso viene eseguito un test relativo unicamente alle funzionalità, in quanto si sottopongono una serie di query, che spaziano dalle interrogazioni alla modifica dei dati, per arrivare a query sintatticamente errate che mettano in luce il trasferimento dell'eccezione tra client e server.

I capitoli precedenti sono stati focalizzati sulle funzionalità correlate alla persistenza, e sul come tali funzionalità potevano essere estese tramite la creazione di librerie software di supporto.

In questo capitolo, si porrà l'accento sull'aspetto prestazionale, innanzitutto caratterizzando il comportamento nativo della piattaforma App Engine, e poi analizzando come l'introduzione del progetto *nembo* comporti una variazione delle prestazioni.

5.1 Gestione delle sessioni

5.1.1 Descrizione

Durante l'analisi dell'architettura di App Engine, abbiamo evidenziato come il codice eseguito all'interno del Runtime Environment debba essere inteso come *stateless*, in quanto non si hanno garanzie che due richieste successive alla stessa servlet possano condividere una qualche forma di stato, a causa delle modalità di gestione delle web request.

Lo standard delle *HttpServlet* prevede però una forma di gestione dello stato, attraverso l'interfaccia *HttpSession*, che consente di gestire una serie di richieste attraverso un'unica sessione, mettendo a disposizione una serie di metodi relativi sia all'identificazione della sessione, sia alla condivisione di dati per le *HTTP request* all'interno di un'unica sessione.

Lo scopo del progetto è verificare come le *HttpSession* sono supportate all'interno della piattaforma App Engine, sviluppando un semplice codice di test e sfruttando i risultati generati per evidenziare l'organizzazione architetturale in questo contesto.

5.1.2 L'interfaccia HttpSession

L'interfaccia *HttpSession* [23] consente la gestione di una sessione all'interno delle servlet: in questo modo è possibile definire un oggetto che sopravvive attraverso richieste HTTP multiple, e che è solitamente correlato ad un unico utente, al quale è possibile collegare altri oggetti (in questo contesto definiti *attributi di sessione*) che saranno disponibili per tutte le connessioni avvenute all'interno di una sessione.

Solitamente la sessione rimane attiva per un periodo di tempo limitato; il modo in cui questa sessione viene stabilita è lasciato alla implementazione dell'HTTP server, e include la generazione di cookie sul client oppure la modifica degli URL utilizzati dalle connessioni.

In un codice minimale di una servlet basato su sessioni, si parte dall'oggetto *HttpServletRequest* collegato alla *HTTP request* che ha attivato la

servlet, dal quale è possibile ottenere un oggetto di tipo `HttpSession` tramite il metodo `getSession`: si noti come tale metodo si occupi di ritornare la sessione associata alla richiesta in corso, e genera una nuova sessione nel caso non ve ne sia una attiva.

Nelle due servlet di prova che utilizzeremo per il nostro test (codice in appendice A.28 e A.29), dopo aver ottenuto l'accesso alla sessione, stampiamo il risultato del metodo `getId`, che ci ritorna appunto l'identificatore della sessione.

Utilizziamo inoltre la prima servlet per salvare un valore all'interno della nostra sessione, e la seconda per andare a rileggerlo e stamparlo a video: in caso di successo del test, saremo in grado di ottenere dalla servlet di `GetAttribute` il valore di un attributo che è stato inizializzato dalla `SetAttribute` e che persiste tra le due *HTTP request* proprio grazie alla sessione.

5.1.3 Supporto su AppEngine

Verificando la documentazione^[24] prendiamo atto che il supporto alle sessioni è disabilitato per default in App Engine, e deve essere esplicitamente abilitato inserendo la sezione

```
<sessions-enabled>true</sessions-enabled>
```

all'interno del file `appengine-web.xml` (vedi appendice A.31). Sono stati modificati anche i files `index.html` e `web.xml` (in appendice A.30 e A.32) in modo da definire entrambe le servlet e presentare un semplice menu che consente all'utente di avviare alternativamente la parte di `SetAttribute` o `GetAttribute`. In questo modo è stato possibile testare il funzionamento delle sessioni direttamente sulla piattaforma di cloud, ottenendo i seguenti risultati per la `SetAttribute`

```
Session id: 5MRxHhDy-yrdfLyL6dasSQ
```

e per la `GetAttribute`

```
Session id: 5MRxHhDy-yrdfLyL6dasSQ  
valore1
```

5.1.4 Osservazioni

Possiamo osservare come il funzionamento delle due servlet corrisponda a quanto descritto dalle specifiche: due *HTTP request* separate condividono la stessa sessione (della quale visualizziamo l'id); inoltre l'attributo *attributo1*, al quale viene assegnato il valore *valore1* dalla `SetAttribute`, può essere recuperato dalla `GetAttribute` che visualizza in output proprio il valore atteso.

A titolo puramente informativo notiamo come la sessione venga implementata nel nostro caso di test, attraverso uno scambio di cookies tra il server App Engine e il browser Chrome, che viene presentato nel successivo screenshot.



Figura 5.1: Cookie utilizzato per la gestione della sessione

Ben più interessante risulta invece il meccanismo mediante il quale vengono realizzate le sessioni in App Engine. Come abbiamo precedentemente evidenziato, finché si rimane all'interno del Runtime Environment la logica è del tipo *stateless*, proprio perchè ciascuna *request* potrebbe essere idealmente gestita da una macchina diversa all'interno del cloud, senza quindi la possibilità di condividere alcun dato in memoria.

Il meccanismo principe per implementare la logica *stateful* in App Engine è il Datastore, che è appunto distribuito, ad accesso condiviso tra le macchine del cloud, dunque risulta logico attendersi che anche la gestione delle sessioni debba coinvolgere tale struttura.

Quando una sessione viene attivata, sul Datastore viene memorizzata una entità del tipo `_ah_SESSION`, con una chiave formata dall'identificatore di sessione al quale si aggiunge il prefisso `_ahs`. Inseriamo uno screenshot del Datastore per una entità di questo tipo.

Tali entità sono formate dai due attributi

`_expires` indica la scadenza della sessione, utilizzando un timestamp in millisecondi: l'impostazione di default prevede una durata di 24 ore, e tale attributo viene aggiornato ad ogni utilizzo della sessione

`_values` è un valore binario (tipo *blob*) che rappresenta una serializzazione dell'HashSet[25] che memorizza gli attributi di sessione, sotto forma di coppie `<chiave, valore>`.

Edit Entity: `_ah_SESSION`

Decoded entity key: `_ah_SESSION: name= ahs5MRxHhDy-yrdflYl6dasSQ`

Entity key: `agl0ZXN0LWNyYWYKwsSC19haF9TRVNTSU9OIhpFYWhzNU1SeEhoRHkteXJkZkx5TDZkYXNTUQw`

Enter information for the entity below. If you'd like to change a property's type, set it to Null, save the entity, edit the entity, or change the type.

`_expires`

value:

type:

`_values`

value: 105 bytes, SHA-1 = `fc7a38db4bbc6a5f82b54392133edadcbec775ef`

type: blob

Figura 5.2: Entità di tipo `_ah_SESSION` memorizzata nel Datastore

Una ottimizzazione di questo meccanismo è rappresentata dall'utilizzo della memcache per ridurre gli accessi al datastore, introducendo un meccanismo di caching locale delle sessioni.

Concludiamo comunque osservando come l'implementazione delle sessioni in App Engine rispetti l'organizzazione architetturale esposta in precedenza, dove ogni *request* viene gestita nel modo il più possibile indipendente dal Runtime Environment: nel caso si voglia introdurre una qualche forma di dato persistente tra due diverse esecuzioni, lo strumento principe è il Datastore distribuito.

5.2 Inserimenti su datastore

5.2.1 Descrizione

Per conoscere le prestazioni delle operazioni di persistenza sul datastore, è stato messo a punto un test che genera un insieme di oggetti che viene successivamente memorizzato sul datastore, all'interno del quale vengono determinati i tempi di esecuzione in diverse sezioni significative, al fine di ottenere delle indicazioni sulle latenze e sulla scalabilità.

Questi test sono costruiti attraverso l'interazione da due progetti Java che verranno descritti in maggiore dettaglio nelle sottosezioni successive.

Come primo passo è stata creata una servlet da eseguire su App Engine, con il compito di generare un insieme di entità e di memorizzarle nel datastore. Tale servlet misura i tempi di esecuzione delle singole operazioni di inserimento – da cui ricava i valori minimi, massimi e medi – oltre al proprio tempo totale di esecuzione, e lo ritorna come output.

Al fine di generare un carico operativo sufficiente a mettere in luce le caratteristiche di scalabilità, viene lanciata una applicazione Java sulla macchina locale. Tale applicazione genera un certo numero di thread, ciascuno dei quali va a richiamare la servlet, determinandone il tempo di risposta e memorizzando i risultati su disco.

Il numero di thread concorrenti viene aumentato secondo una progressione geometrica, in modo da verificare la risposta del sistema al crescere del carico: in base all'analisi dei risultati verranno proposti una serie di grafici corredati da osservazioni.

La crescita del numero di richieste concorrenti alla servlet secondo una progressione geometrica è supportata dalla necessità del *frontend* App Engine di attivare un numero di istanze sufficienti a supportare le richieste generate: la generazione immediata di un grosso pool di connessioni concorrente provocherebbe un alto tasso di richieste che non vengono soddisfatte, a causa del tempo insufficiente per l'attivazione di un congruo numero di istanze.

Per maggiori dettagli sui meccanismi di scalabilità in App Engine, si rimanda alla recente sessione al Google I/O 2011¹.

5.2.2 gae-time-datastore

Questo progetto definisce la servlet che viene attivata su App Engine, in modo da generare il carico operativo su cui viene osservata la risposta del sistema in termini di prestazioni e scalabilità.

Come punto di partenza si è preso il precedente test-crud-single (in sezione 3.1), che andava a generare una singola entità per poi testare tutte le operazioni previste dal modello CRUD. E' stata rinominata l'entità Progetto, trasformandola in Record (codice in appendice A.34), al semplice scopo di semplificare la stesura della descrizione del test.

La parte qualificante del progetto è la servlet (vedi appendice A.35), che va a generare e memorizzare su persistenza L record. Per ogni operazione di inserimento viene determinato il tempo impiegato, in millisecondi. Alla fine del ciclo su L record, vengono determinati i valori massimo, minimo e medio del tempo di inserimento, oltre al tempo totale di esecuzione della servlet, che vengono ritornata all'interno della *HTTP response* da restituire al client.

Come descritto nella sezione 2.2.3, il meccanismo di *app caching* prevede che le sezioni *static* vengano eseguite solo durante il caricamento della app, in modo da mantenere in memoria l'istanza e riutilizzarla per soddisfare le richieste successive: grazie a questo meccanismo, è stato possibile

¹Il Google I/O è una conferenza annuale destinata agli sviluppatori su tecnologie Google nei settori web, mobile ed enterprise: è stata introdotta nel 2008 e si svolge solitamente nello spazio di due giornate a San Francisco.

definire una sorta di *id di istanza* che verrà successivamente utilizzato per verificare il meccanismo di attivazione.

Nei primi test, si è osservato come al crescere della concorrenza si verificavano degli errori negli inserimenti, che causavano la terminazione prematura del ciclo: dopo una verifica della documentazione e del codice, si è osservato come l'accesso all'*EntityManager* non veniva gestito in modo *thread-safe*, e questo poteva dare origine a problemi di concorrenza.

Per ovviare a questo inconveniente, è stato aggiunto lo specificatore *synchronized* ai metodi della classe *CrudProviderImpl* (in appendice A.33) in modo da garantire il corretto accesso alle sezioni critiche.

5.2.3 local-time-datastore

Il secondo progetto creato si configura come una applicazione Java, che viene lanciata sulla macchina locale. Tale applicazione esegue una serie di test, ciascuno dei quali comprende K chiamate alla servlet definita nella sezione precedente.

Ciascun test gestisce le chiamate secondo un grado di parallelismo P , che viene progressivamente aumentato seguendo una progressione geometrica: il primo ciclo prevede la generazione delle richieste secondo un meccanismo sequenziale, il secondo prevede l'esistenza di 2 chiamate attive in modo simultaneo, il terzo ne utilizza 4, fino ad arrivare al limite di 512.

Tale limite è determinato da due considerazioni:

- l'esecuzione del test con $L=100$ e $K=500$, dove P va a sfruttare tutta la progressione geometrica da 1 a 512, esaurisce quasi completamente la quota giornaliera di *CPU time* disponibile come quota gratuita;
- sul sistema Linux all'interno del quale viene eseguito il progetto, il numero massimo di connessioni HTTP che possono essere contemporaneamente aperte è limitato dal numero di descrittori di file che viene limitato dal kernel, e già nel passo con $P=1024$ compaiono delle eccezioni che segnalano questo inconveniente. Sarebbe possibile evitare questo inconveniente incrementando il limite imposto dal kernel per i descrittori esistenti, ma al momento tale opzione non viene praticata.

La classe *Launcher* (in appendice A.37), che viene attivata dal metodo *main* (appendice A.36) va a costruire le singole chiamate alla servlet, gestendo sia la progressione su P che il contatore su K : è interessante osservare come si faccia uso di un semaforo per limitare il grado di parallelismo.

All'inizio di un test, vengono lanciate in contemporanea P richieste: dopodiché, grazie alla sincronizzazione indotta dal semaforo, ogni risposta ad una attivazione della servlet abilita una nuova richiesta, fino ad esaurire il budget imposto da K . In questo modo, abbiamo sempre P richieste pendenti sulla servlet *gae-time-datastore*.

La gestione della singola richiesta viene demandata alla classe `Single-Test` (in appendice A.38), che si occupa appunto di sincronizzarsi con il semaforo, di generare la *HTTP request* e di riportarne i risultati su file, determinando inoltre il tempo di risposta dal lato client.

I risultati vengono memorizzati da un file CSV² che può essere facilmente rielaborato come un semplice foglio di calcolo, in modo da generare dei grafici riassuntivi.

5.2.4 I test effettuati

Una prima idea di test consisteva nel far crescere gradualmente il database attraverso una serie di chiamate più o meno concorrenti alla servlet, per verificare se esisteva una qualche correlazione tra la dimensione della tabella e il tempo di inserimento del record.

Tale test è stato successivamente abbandonato, in quanto, come si può osservare in figura 5.3, le latenze all'interno di una singola giornata sono estremamente variabili, dunque risulta molto difficile separare la variabilità intrinseca dall'effetto indotto dalle problematiche di scalabilità. Si è

Put: Latency

Measures the latency, in milliseconds, of a call to `datastore.put()` on a single entity containing 2kB of data.

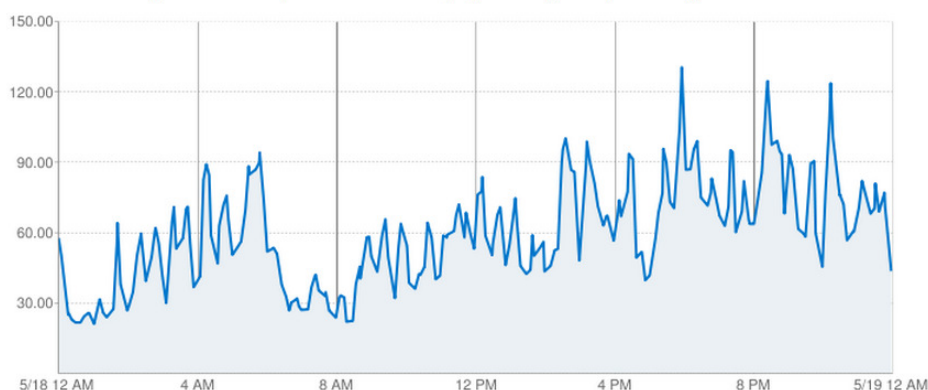


Figura 5.3: Variazione della latenza di inserimento del Datastore

quindi provveduto a collezionare i dati relativi all'esecuzione dei test con particolare attenzione ai valori massimi, minimi e medi in modo non correlato alla scalabilità, in modo da ottenere delle osservazioni sulla stabilità delle latenze in relazione al tempo, con delle indicazioni anche sul caso pessimo e sul caso ottimo.

²Il `COMMA-SEPARATED VALUES` è un formato per l'esportazione di dati tabulari all'interno di files di testo. Non è uno standard, ma solo una serie di prassi consolidate che determinano la presentazione in righe, separate da *separatori di riga*, dei record che a loro volta sono formati da campi separati da *separatori di campo*.

Preziose indicazioni sono venute dal confronto dei tempi di esecuzione totali, rilevati sia lato server (all'interno servlet), sia lato client (ossia il tempo che intercorre tra l'invio della *HTTP request* e la ricezione della risposta).

Il parametro L è stato variato nei test prevedendo i tre valori 1, 10, 100 attraverso successivi *redeploying* su App Engine, in modo da testare la variabilità dei tempi e soprattutto la risposta del sistema in termini di generazione di nuove istanze.

5.2.5 Osservazioni sulle latenze di inserimento

Nei grafici da B.1 a B.5 e da B.11 a B.14 vengono presentati una serie di grafici ritenuti significativi, fissando $L \in \{100, 10\}$ e $P \in \{1, 8, 64, 512\}$.

La suite di test è stata invece portata a termine per $L \in \{100, 10, 1\}$ e $P \in \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512\}$, ma i risultati non vengono allegati in quanto genererebbero un volume eccessivo di dati.

Per quanto riguarda le latenze di inserimento, possiamo notare che:

- i valori minimi sono di circa 15 ms, dove il dato migliore si attesta sugli 11 ms;
- i valori massimi hanno una grande variabilità, con alta probabilità si rimane all'interno del centinaio di ms; si sono tuttavia verificati casi peculiari in cui il ritardo è dell'ordine dei secondi, con un picco prossimo ai 10 s;
- il valore medio del tempo di inserimento è stimabile in circa 25 ms;
- non è stato possibile riscontrare marcate differenze nei tempi che possano indicare una qualche correlazione con la dimensione del dataset; allo stesso tempo, si è osservato come l'andamento medio dei tempi rifletta le latenze che vengono visualizzate dai grafici del Pannello di Amministrazione di App Engine.

E' possibile osservare un interessante fenomeno nel grafico B.5, dove vengono presentati le latenze raggruppate per *id di istanza*: la distribuzione dei valori, soprattutto relativa al valore minimo e al valore medio, sembra suggerire che tali misure sono in qualche modo legate all'effettiva istanza della servlet su cui sono determinate.

In altri termini, il grafico sembra suggerire come nello stesso momento diverse istanze della stessa app sperimentino tempi diversi nell'accesso al datastore. Nella documentazione non è stato possibile approfondire tale fenomeno, si può però ipotizzare che tale comportamento sia da imputare al carico istantaneo della macchina su cui l'istanza è in esecuzione o su latenze indotte dalla topologia del cloud, dove gli app server devono dialogare con i server di datastore per la gestione della persistenza.

5.2.6 Osservazioni sui tempi di risposta della servlet

In questa sezione, ci concentreremo sul confronto tra il confronto tra i tempi di esecuzione della servlet, che possiamo definire in due modi:

lato server come il tempo trascorso tra l'avvio dell'elaborazione della servlet e la chiusura della stessa, che coincide con la resituzione della *HTTP response*;

lato client come il tempo trascorso tra l'invio della *HTTP request* e la ricezione della risposta.

In una prima osservazione, dovrebbe valere che $\text{tempo lato client} = \text{tempo lato server} + \text{RTT}$, dove $\text{RTT} = \text{Round Trip Time}$ è il tempo impiegato per il viaggio di un pacchetto di dimensione trascurabile dalla macchina locale al frontend di App Engine, ritorno compreso.

La prima complicazione a questa assunzione è dovuta al fatto che nella topologia della rete da noi utilizzata esistono molti punti in cui potrebbero intervenire rallentamenti tali da introdurre dei ritardi. In questo senso possiamo ritenere trascurabili le variazioni nell'RTT introdotte nell'attraversamento dei router della rete Internet.

La connessione alla rete del laboratorio su cui sono stati condotti i test sfrutta un meccanismo di *transparent proxy*³: tale componente rappresenta una potenziale fonte di ritardo nell'RTT.

Come è possibile osservare nel grafico B.19, in questo test il tempo di risposta lato server ha avuto una durata media intorno ai 300 ms, e la maggior parte dei valori sono compresi tra i 200 e i 500 ms. Viceversa, dal lato client si può assistere ad un comportamento anomalo: una prima serie di risposte arriva con tempi che crescono linearmente fino a circa 2,2 s (il motivo di questa crescita verrà presentato nei prossimi paragrafi).

Poi assistiamo ad un cluster di risultati che arrivano praticamente in contemporanea, a circa 4,2 s, dopodiché il tempo torna a salire linearmente. È interessante osservare come il primo e il terzo segmento siano assimilabili ad una stessa retta (si confronti con il grafico B.14 in cui la perturbazione introdotta dal proxy è trascurabile).

L'anomalia riguarda quindi il cluster di risposte che vengono riportate in contemporanea, senza evidenziare fenomeni di ritardo dal lato server: una plausibile spiegazione è data da un ritardo introdotto appunto dal meccanismo di proxy, in cui le risposte del server App Engine incorrono in un meccanismo di caching prima di essere trasmesse al nostro client.

Si tratta di un comportamento estremamente caratteristico ed evidente nella lettura dei dati, si è quindi provveduto a ripetere il test quando in fase

³Il *transparent proxy* è una architettura introdotta per la navigazione, quindi a livello HTTP, in cui il proxy è un componente intermedio attraverso il quale vengono instradate tutte le connessioni HTTP e ne gestisce il monitoraggio, il caching e il controllo.

di analisi dei risultati è emerso il sospetto che l'RTT fosse stato contaminato da effetti spuri indotti dal proxy.

Nei grafici da B.6 a B.10 e da B.15 a B.18 sono quindi presentati grafici significativi, che mettono a confronto i tempi di risposta misurati come precedentemente descritto, e sfruttano gli stessi settaggi dei parametri della sottosezione precedente.

Per quanto riguarda le misurazioni lato server, il comportamento è in linea con le nostre previsioni, in quanto il tempo di esecuzione della singola servlet è essenzialmente dovuto al tempo medio di inserimento della singola entità, moltiplicato per il coefficiente L che definisce il numero di inserimento da compiere.

Al contrario, il comportamento delle tempistiche lato client suggerisce molte osservazioni:

- se escludiamo perturbazioni indotte dal proxy, la differenza tra il tempo di risposta lato server e lato client è dovuto alla somma di due fattori: l'RTT e le latenze dovute al frontend
- nei casi dove $P=1$, le latenze dovute al frontend sembrano trascurabili, possiamo quindi stimare un RTT intorno ai 500 ms;
- man mano che il parametro P viene incrementato, la latenza introdotta nel frontend cresce di conseguenza fino a diventare preponderante, in quanto l'RTT ha una variabilità limitata (*jitter*) e indipendente dal numero di istanze attivate;
- il frontend reagisce all'aumento del carico attivando nuove istanze: le attivazioni seguono però dei criteri che regolano il meccanismo di scalabilità, valutando caso per caso tra l'attivazione una nuova istanza o l'accodamento della richiesta su una istanza esistente;
- per ciascuna istanza attiva, viene quindi popolata una coda di richieste pendenti che vengono elaborate in modo sequenziale: tale comportamento è particolarmente evidente nel grafico B.10, dove il tempo di risposta è ordinato per *id di istanza* relativamente ai parametri $P=512$ e $L=100$;
- nei grafici B.9 e B.18, dove $P=512$, è evidente come il pool di istanze contribuisca a esaurire il blocco di richieste contemporanee, attraverso un accodamento sequenziale che beneficia però di un grado di parallelismo dato dal numero di istanze attivate.

5.2.7 Osservazioni sulle istanze attivate

La tabella 5.1 e il grafico 5.4 mettono in relazione il numero di istanze attivate con i parametri P e L che vengono fatti variare all'interno dei test.

Vengono quindi presentate tre serie dati, relative alle tre opzioni 1, 10, 100 su cui è stato testato L . Per ciascuna serie, il parametro P spazia in progressione geometrica da 1 a 512, e per ciascun caso è stato determinato il numero di istanze attivate da App Engine in risposta al pool di richieste.

In particolare osserviamo come il numero di attivazioni cresca in modo graduale per adattarsi alla crescita del numero di richieste pendenti, fino ad arrivare ad un massimo di 29.

Il massimo delle attivazioni si è ottenuto per $L=10$, in quanto

- per $L=1$ non viene generato un carico sufficiente ad attivare completamente il meccanismo scalabilità, in quanto il tempo massimo di risposta misurato nel picco di carico è inferiore ai 10 s.
- per $L=100$ il carico è sufficiente a sollecitare il meccanismo di attivazione, tanto che il tempo massimo di risposta è vicino ai 65 s; tuttavia la servlet ha un tempo di esecuzione di 2,5 s, non ottimale in quanto per ottenere il massimo della scalabilità tale tempo dovrebbe essere inferiore al secondo.

Per $L=10$ il carico generato è sufficiente ad attivare un buon numero di istanze, inoltre il tempo di risposta medio è di circa 0,3 s che si adatta meglio ai meccanismi previsti dall'*app caching*.

E' interessante osservare come il tempo massimo di risposta nel caso $L=100$ e $P=512$ sia di circa 65 s: ipotizzando che le 500 richieste contemporanee vengano servite da 22 istanze in parallelo, e che il tempo medio di risposta della singola app è di circa 2,5 s, si avrebbe un tempo totale di esecuzione di $\frac{2,5 \text{ s} * 500 \text{ richieste}}{22 \text{ istanze}} \approx 57 \text{ s}$, che si avvicina al valore osservato.

Tale risultato indica come le risorse del cloud vengano intensamente sfruttate da questa casistica, e le attivazioni di istanze siano limitate unicamente dai criteri del meccanismo di *app caching*.

5.3 Inserimenti su *nembo*

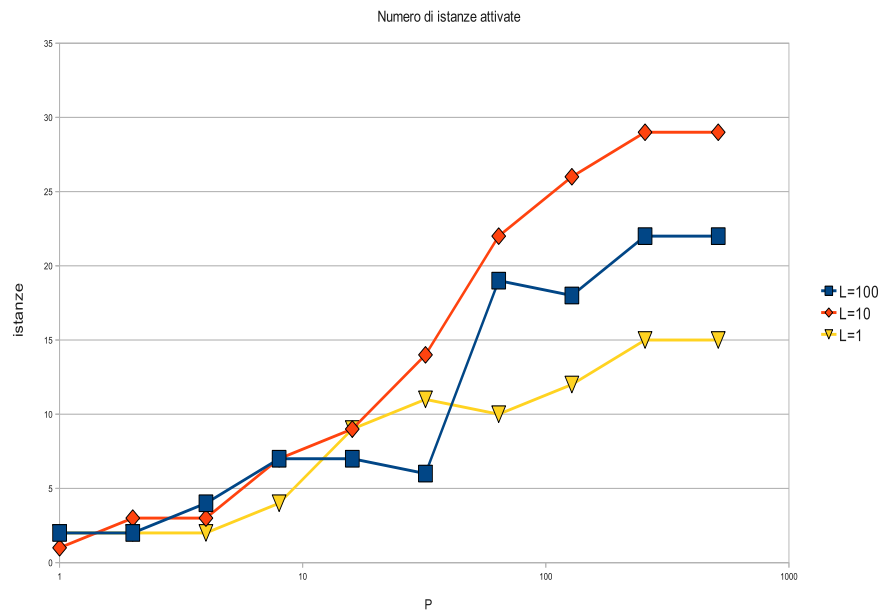
Oltre alla verifica funzionale del progetto, si è pensato di condurre una serie di test prestazionali, in modo da verificare come le latenze di accesso alla persistenza vengono influenzate dall'utilizzo di un web service esterno al cloud, e come varia la scalabilità dell'applicazione di test all'interno runtime environment di App Engine.

L'architettura della batteria di test è estremamente simile ai test condotti sugli *Inserimenti sul datastore* (si veda la sezione 5.2): anche in questo caso il carico operativo viene generato dall'interazione tra una servlet ospitata su App Engine e una applicazione locale che riproduce una serie di chiamate a parallelismo crescente.

Presenteremo brevemente i due progetti utilizzati, evidenziando come sono stati modificati rispetto ai test su *datastore*.

Tabella 5.1: Attivazione di istanze in relazione a P e L

K	$L=100$	$L=10$	$L=1$
1	2	1	2
2	2	3	2
4	4	3	2
8	7	7	4
16	7	9	9
32	6	14	11
64	19	22	10
128	18	26	12
256	22	29	15
512	22	29	15

Figura 5.4: Attivazione di istanze in relazione a P e L

5.3.1 gae-time-nembo

Questo progetto è basato su *gae-time-datastore*: in modo analogo, va a creare una serie di L record per poi memorizzarli su persistenza. La differenza sostanziale è che questi record non sono più memorizzati all'interno del

datastore, in quanto ci si appoggia alla libreria *nembo-gae* che, come ricorderemo, gestisce la persistenza su un DBMS relazionale esterno ad App Engine.

Sono stati condotti test con i valori di $L \in \{100, 10, 1\}$, analogamente a quanto condotto sul datastore, in modo da poter stimare sia la variazione delle latenze di inserimento (che sicuramente andranno ad aumentare, in quanto la persistenza è realizzata all'esterno del cloud App Engine) sia il mantenimento di una buona scalabilità.

5.3.2 local-time-nembo

Anche in questo caso, il progetto ricalca fortemente la struttura di *local-time-datastore*, generando un pool di richieste a *gae-time-nembo*, con un grado di parallelismo crescente secondo una progressione geometrica. Dato che la risposta dell'applicazione web richiederà certamente un tempo maggiore, e comporterà un maggiore dispendio di risorse computazionali, il grado di parallelismo che andremo a testare viene limitato.

In quest'ottica, è stato possibile utilizzare $K=256$ e P che sfrutta la progressione geometrica da 1 a 256 in tutti i test, tranne il caso in cui la servlet utilizza $L=100$ (in cui il tempo di esecuzione è maggiore), in cui K è stato limitato a 128 e la progressione geometrica non ha previsto il caso $P=256$.

5.3.3 Osservazioni sulle latenze di inserimento

La struttura di *nembo* prevede che, per ogni operazione di accesso al database, venga realizzata una richiesta web al servizio *nembo-db*: tale servizio si trova certamente al di fuori di Google App Engine (nei nostri è ospitato da una macchina all'interno del servizio EC2 sul cloud Amazon).

Sotto questa ipotesi, è ragionevole aspettarsi che le latenze subiscano un aumento, proprio per la necessità di costruire una *web request* che deve viaggiare sulla rete internet, e non può quindi beneficiare appieno della velocità della rete interna che il cloud Google può vantare.

Dopo aver condotto i test, possiamo notare che:

- i valori minimi si attestano intorno agli 85-90 ms, con un valore minimo vicino agli 80 ms;
- i valori massimi risultano certamente più variabili, spaziando nella maggioranza dei casi in una finestra compresa tra i 150 e i 500 ms, anche se in casi peculiari superiori a 1s;
- il valore medio del tempo di inserimento su *nembo* è stimabile in circa 105 ms;

Come previsto, le latenze di inserimento hanno subito un peggioramento sensibile, anche se inferiore alle aspettative: è importante evidenziare

come tale peggioramento non sia direttamente connesso con il passaggio a un DBMS relazionale, viceversa debba essere imputato all'architettura del progetto.

La scelta di modellare il servizio di persistenza come un web service, e soprattutto la necessità di ospitare questo servizio presso una infrastruttura esterna al cloud Google, comporta l'introduzione di latenze dovute al passaggio di pacchetti in reti a velocità non massimale che separano i due sistemi.

5.3.4 Osservazioni sui tempi di risposta della servlet

Sotto questo punto di vista, il comportamento delle servlet di test non ha evidenziato differenze rispetto al caso relativo al datastore. Si tratta di un risultato che era auspicabile, in quanto tale metrica coinvolge unicamente il front-end di App Engine, e il modo in cui le istanze dell'applicazione vengono attivate al crescere del carico.

Da questo punto di vista, l'unica differenza tra una applicazione che memorizzi dati su datastore, e una che li memorizzi su *nembo*, è riconducibile a un maggiore tempo di esecuzione della seconda proprio a causa del crescere delle latenze.

Sotto questa ipotesi, riconosciamo come la realizzazione dei meccanismi di scalabilità in App Engine confermi la sua bontà, consentendo una gestione efficiente e trasparente della scalabilità delle applicazioni che vi sono ospitate.

5.3.5 Osservazioni sulle istanze attivate

Proponiamo ora un grafico analogo a quanto realizzato nel test precedente, al fine di mettere in relazione il numero di istanze attivate con la variazione dei parametri all'interno del test.

Osserviamo come l'andamento delle curve non subisca particolari variazioni, e confermi le ipotesi relative alla gestione della scalabilità fin qui avanzate.

E' importante notare come il numero di istanze attivate risulti significativamente maggiore rispetto al caso precedente: pur con un grado di parallelismo leggermente minore, il numero di istanze attivate viene mediamente raddoppiato.

Dato che i test prestazionali relativi a *nembo* sono stati eseguiti ad alcuni mesi di distanza dai test relativi al datastore, si è ipotizzato che le logiche che comandano l'attivazione delle istanze avessero subito una variazione e quindi la batteria di test precedenti fosse da ripetere.

Dopo la ripetizione dei test relativi al datastore, si è potuto notare che effettivamente il numero di istanze attivate con carichi elevati ha subito un incremento, stimabile mediamente in un 20%.

Questo ci suggerisce una modifica nel meccanismo di attivazione, probabilmente dovuto alla crescita dell'infrastruttura del cloud che ha permesso un approccio più aggressivo nell'attivazione delle istanze.

Allo stesso tempo, si può evidenziare come il numero di istanze attivate durante i test su *nembo* risulti comunque notevole anche sotto questa ipotesi: ne possiamo dedurre che il carico operativo generato da *nembo* è maggiore proprio per le latenze più elevate e la necessità di sfruttare un web service esterno.

Tale carico non ha però comportato un peggioramento sensibile nelle latenze lato client, come evidenziato in precedenza: questo risultato conferma come l'architettura di App Engine si adatti efficacemente all'accresciuta richiesta di risorse, allocando più risorse in modo da garantire buone prestazioni.

Tabella 5.2: Attivazione di istanze in relazione a P e L

K	$L=100$	$L=10$	$L=1$
1	1	1	1
2	2	2	2
4	3	3	3
8	4	4	6
16	5	10	12
32	13	24	21
64	20	43	31
128	22	48	41
256		52	45

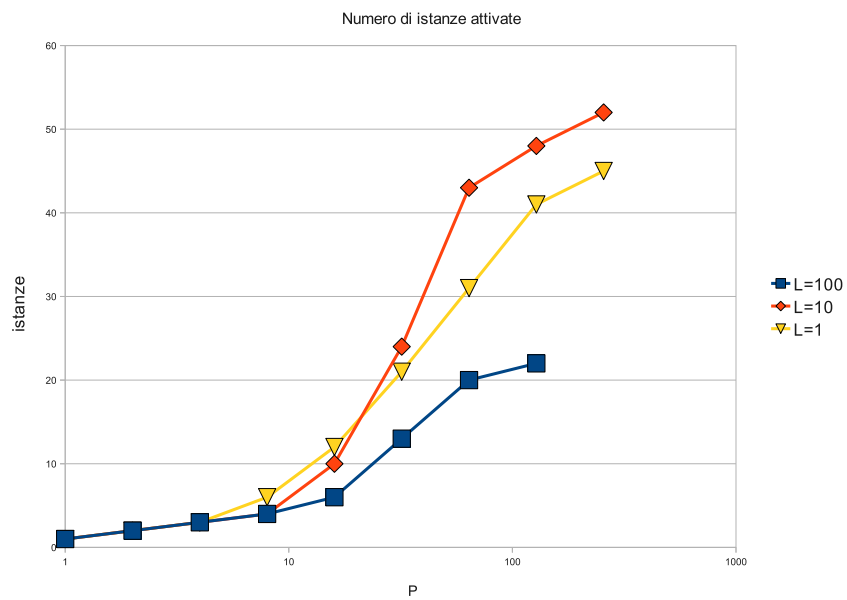


Figura 5.5: Attivazione di istanze in relazione a P e L

In questa tesi, abbiamo potuto evidenziare come la piattaforma di cloud computing proposta da Google risulti certamente una proposta degna di attenzione per la sua innovatività e per i servizi che è in grado di supportare.

Il trend dello sviluppo di applicazioni web su piattaforma rappresenta ad oggi una realtà ben affermata, che consente agli sviluppatori l'implementazione e l'utilizzo delle logiche necessarie al proprio software, con un approccio ad alto livello in cui molte funzionalità vengono demandate alla piattaforma sottostante.

Google App Engine consente di trasportare questo paradigma nel mondo del cloud computing, offrendo così un servizio on-demand con potenzialità di scalabilità e affidabilità in grado di supportare la crescita della base d'utenza degli applicativi.

Per quanto riguarda la realizzazione di applicazioni stateless, le limitazioni imposte sono minime, e in certi casi sono omogenee a quanto viene comunque imposto nella generica programmazione su piattaforma (pensiamo al limite nel tempo di esecuzione della servlet o nel divieto di generare thread concorrenti).

Allo stesso tempo, l'incapsulamento di alcune funzionalità all'interno di specifici servizi della piattaforma (si pensi all'URL Fetch che consente la generazione di connessioni HTTP verso servizi web esterni) risulta comunque di immediata gestibilità, in quanto la sua gestione è trasparente allo sviluppatore.

Nel momento in cui si va ad introdurre l'accesso alla persistenza, la situazione cambia radicalmente, per la quantità e la complessità dei vicoli che vengono introdotti, se confrontati con il trend comune che prevede l'utilizzo di DBMS relazionali in accoppiamento alle applicazioni.

L'origine di questi vincoli può essere ricondotta alla scelta di non realizzare una persistenza basata su un DBMS relazionale, bensì di sfruttare BigTable, la piattaforma proprietaria realizzata internamente, proponendone un'adattamento alle necessità di App Engine.

In questo modo, uno sviluppatore tradizionale che si avvicini ad App Engine, deve scontrarsi con la mancanza di alcune funzionalità strettamente connesse con il modello relazionale, che abitualmente si trova ad utilizzare. Allo stesso tempo, anche nel caso sia consapevole della diversa modalità di gestione di una base di dati non relazionale, e abbia comunque esperienza con uno strumento come JPA, lo sviluppatore dovrà comunque confrontarsi con vincoli progettuali a volte anche molto stringenti.

Tali vincoli progettuali derivano direttamente dalla struttura architetturale di BigTable, e hanno il preciso scopo di assicurare alte prestazioni e scalabilità anche nelle situazioni di massimo carico: la piattaforma è quin-

di ottimizzata per applicazioni web a basso tempo di risposta e ad elevati vincoli di scalabilità.

Sotto quest'ottica, anche i test da noi effettuati confermano che una applicazione realizzata ex-novo, che rientri in questa definizione, può essere realizzata con relativa semplicità, proprio per la capacità del sistema di inibire allo sviluppatore la costruzione di logiche che possano inficiare la scalabilità del prodotto.

Lo sviluppo diventa certamente più problematico quando ci si propone di riutilizzare del codice, comunque scritto sotto il paradigma delle applicazioni su piattaforma: una frazione non banale degli sviluppatori potrebbe trovare il proprio proposito irrealizzabile, a causa della mancanza di una base di dati relazionale, che potrebbe essere richiesta dal codice preesistente.

Anche nell'ipotesi che il codice possa supportare la mancanza del modello relazionale, lo sviluppatore dovrà confrontarsi con molte limitazioni che rischiano di obbligarlo alla riscrittura completa di certe sezioni del codice.

Nei test da noi realizzati, abbiamo dimostrato come queste limitazioni funzionali possano essere superate con l'introduzione di un servizio di persistenza, parallelo al datastore già esistente, basato su un DBMS relazionale e con pieno supporto delle sue API di accesso.

I nostri test hanno certamente evidenziato un peggioramento delle prestazioni, relativamente alle latenze d'accesso e alle potenzialità di scalabilità, anche se occorre evidenziare come una ingegnerizzazione più spinta del codice, ma soprattutto l'utilizzo di infrastrutture interne al cloud App Engine, potrebbero mitigare in modo sensibile questa differenza.

E' plausibile che il prodotto relazionale avrà comunque prestazioni inferiori al datastore esistenze, soprattutto sulle metriche relative alla scalabilità pura, proprio perchè l'introduzione di funzionalità aggiuntive comporta la necessità di utilizzare strutture che non possono garantire scalabilità assoluta.

Possiamo evidenziare come una soluzione concorrente, ossia Microsoft Windows Azure Platform, che in molti aspetti risulta simile ad App Engine, metta invece a disposizione dei propri utenti un DBMS relazionale, basato sulle proprie tecnologie.

Questo conferma l'assunto che l'introduzione di una persistenza su modello relazionale all'interno di una infrastruttura di cloud ha una *realizzabilità tecnologia* certa.

Certamente, tale soluzione comporta problematiche evidenti quando ci si confronta con vincoli elevati di scalabilità, e sotto quest'ottica il datastore Google dimostra la propria superiorità.

La scelta strategica di Google, relativamente alla persistenza, a nostro avviso ha *motivazioni prestazionali* innegabili: ci sentiamo però di evidenziare come tale orientamento abbia forti limiti dal punto di vista commer-

ziale, in quanto rischia di compromettere l'accesso a una fetta di mercato, di dimensioni considerevoli, che considera il supporto relazionale come una caratteristica irrinunciabile.

Sotto quest'ottica, è auspicabile che Google implementi, parallelamente al datastore, anche un servizio alternativo di persistenza con maggiori funzionalità, anche al costo di un peggioramento prestazionale: sarà poi facoltà dello sviluppatore, in relazione alle caratteristiche dell'applicazione che sta realizzando, la scelta dello strumento più adatto a supportare la persistenza dei propri dati.

A.1 test-crud-single

Listing A.1: Main.java

```
import java.io.PrintWriter;
import main.EntityGenerator;
import dao.CrudProviderImpl;

public class Main {
    public static void main(String[] args) {
        new EntityGenerator(new PrintWriter(System.out,true),
            CrudProviderImpl.Provider.HIBERNATE).doTests();
    } //main
} //Main
```

Listing A.2: CrudProvider.java

```
package dao;

public interface CrudProvider<T,K>{
    public void create(T object);
    public T retrieve(Class<T> c,K key);
    public T update(T object);
    public void destroy(T object);
} //CrudProvider
```

Listing A.3: CrudProviderImpl.java

```
package dao;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;

public class CrudProviderImpl<T, K> implements CrudProvider<T,
    K> {
    private static EntityManagerFactory emf;
    private static EntityManager em;
    public static enum Provider{ECLIPSE_LINK,HIBERNATE,APP_ENGINE
    };

    public CrudProviderImpl(Provider p){
        emf = Persistence.createEntityManagerFactory(p.toString());
        em = emf.createEntityManager();
    } //costr.
```

```
@Override
public void create(T object) {
    EntityTransaction t=em.getTransaction();
    try{
        t.begin();
        em.persist(object);
        t.commit();
    }catch(Exception e){
        e.printStackTrace();
        t.rollback();
    }//catch
}//create

@Override
public T retrieve(Class<T> c,K key) {
    return (T)em.find(c,key);
}//retrieve

@Override
public T update(T object) {
    EntityTransaction t=em.getTransaction();
    try{
        t.begin();
        T aggiornato=em.merge(object);
        t.commit();
        return aggiornato;
    }catch(Exception e){
        e.printStackTrace();
        t.rollback();
        return null;
    }//catch
}//update

@Override
public void destroy(T object) {
    EntityTransaction t=em.getTransaction();
    try{
        t.begin();
        em.remove(object);
        t.commit();
    }catch(Exception e){
        e.printStackTrace();
        t.rollback();
    }//catch
}//destroy
public void close(){
    em.close();
    emf.close();
}//close
}//CrudProviderImpl
```

```
package entities;

import java.io.Serializable;
import java.lang.Long;
import java.lang.String;
import javax.persistence.*;

@Entity
@Table(name = "progettiS")
public class Progetto implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String descrizione;
    @Transient
    public Object[] jdoDetachedState;
    private static final long serialVersionUID = 1L;

    public Progetto() {
        super();
    } //costr

    public Long getId() {
        return this.id;
    } //getId
    public void setId(Long id) {
        this.id = id;
    } //setId

    public String getDescrizione() {
        return this.descrizione;
    } //getDescrizione
    public void setDescrizione(String descrizione) {
        this.descrizione = descrizione;
    } //setDescrizione

    @Override
    public String toString() {
        return "Progetto [id=" + id + ", description=" +
            descrizione + ", hash=" + this.hashCode() + "]";
    } //toString
} //Progetto
```

Listing A.5: MainServlet.java

```
package main;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.http.*;
import dao.CrudProviderImpl;

@SuppressWarnings("serial")
```

```

public class MainServlet extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse
        resp) throws IOException {
        resp.setContentType("text/plain");
        new EntityGenerator(new PrintWriter(resp.getWriter()), true),
            CrudProviderImpl.Provider.APP_ENGINE).doTests();
    } //doGet
} //MainServlet

```

Listing A.6: EntityGenerator.java

```

package main;

import java.io.PrintWriter;
import dao.CrudProviderImpl;
import entities.Progetto;

public class EntityGenerator {
    private PrintWriter out;
    private CrudProviderImpl.Provider provider;
    private CrudProviderImpl<Progetto, Long> crud;

    public EntityGenerator(PrintWriter out, CrudProviderImpl.
        Provider provider){
        this.out=out;
        this.provider=provider;
        crud=new CrudProviderImpl<Progetto, Long>(provider);
    } //costr.

    public void doTests(){
        out.println("Inizio Test tabella singola: "+provider.
            toString());
        Long id=create(crud, out);
        retrieve(crud, id);
        update(crud, id);
        //destroy(crud, id);
        out.println("Fine Test");
        crud.close();
    } //doTests

    private Long create (CrudProviderImpl<Progetto, Long> crud,
        PrintWriter out){
        out.print("C ");
        Progetto p=new Progetto();
        p.setDescrizione("Nuova");
        crud.create(p);
        out.println(p);
        return p.getId();
    } //create

    private void retrieve (CrudProviderImpl<Progetto, Long> crud,
        Long id){
        out.print("R ");
    }

```

```

        Progetto p=crud.retrieve(Progetto.class,id);
        out.println(p);
    }//retrieve

    private void update (CrudProviderImpl<Progetto,Long> crud,
        Long id){
        out.print("U ");
        Progetto p=new Progetto();
        p.setId(id);
        p.setDescrizione("Modificata");
        crud.update(p);
        out.println(p);
    }//update

    private void destroy (CrudProviderImpl<Progetto,Long> crud,
        Long id){
        out.print("D ");
        Progetto p=crud.retrieve(Progetto.class,id);
        if(p!=null){
            crud.destroy(p);
            out.println(p);
        }else
            out.println("Non trovo l'oggetto da eliminare");
    }//destroy
} //TestSingleTable

```

Listing A.7: persistence.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/
  persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
  instance" xsi:schemaLocation="http://java.sun.com/xml/ns/
  persistence http://java.sun.com/xml/ns/persistence/
  persistence_2_0.xsd">
  <persistence-unit name="ECLIPSE_LINK" transaction-type="
    RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</
    provider>
    <class>entities.Progetto</class>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:
        postgresql:progetti"/>
      <property name="javax.persistence.jdbc.user" value="
        postgres"/>
      <property name="javax.persistence.jdbc.password" value="
        torrenera"/>
      <property name="javax.persistence.jdbc.driver" value="org
        .postgresql.Driver"/>
    </properties>
  </persistence-unit>
  <persistence-unit name="HIBERNATE" transaction-type="
    RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>

```

```

<class>entities.Progetto</class>
<properties>
  <property name="javax.persistence.jdbc.url" value="jdbc:
    postgresql:progetti"/>
  <property name="javax.persistence.jdbc.user" value="
    postgres"/>
  <property name="javax.persistence.jdbc.password" value="
    torrenera"/>
  <property name="javax.persistence.jdbc.driver" value="org
    .postgresql.Driver"/>
</properties>
</persistence-unit>
<persistence-unit name="APP_ENGINE">
  <provider>org.datanucleus.store.appengine.jpa.
    DatastorePersistenceProvider</provider>
<class>entities.Progetto</class>
<properties>
  <property name="datanucleus.NontransactionalRead" value="
    true"/>
  <property name="datanucleus.NontransactionalWrite" value
    ="true"/>
  <property name="datanucleus.ConnectionURL" value="
    appengine"/>
</properties>
</persistence-unit>
</persistence>

```

Listing A.8: appengine-web.xml

```

<?xml version="1.0" encoding="utf-8"?>
<appengine-web-app xmlns="http://appengine.google.com/ns/1.0">
  <application>test-crud</application>
  <version>1</version>

  <!-- Configure java.util.logging -->
  <system-properties>
    <property name="java.util.logging.config.file" value="WEB-
      INF/logging.properties"/>
    <property name="appengine.orm.disable.duplicate.emf.
      exception" value="true"/>
  </system-properties>

</appengine-web-app>

```

Listing A.9: web.xml

```

<?xml version="1.0" encoding="utf-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee

```



```
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version
="2.5">
<servlet>
  <servlet-name>Main</servlet-name>
  <servlet-class>main.MainServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>Main</servlet-name>
  <url-pattern>/main</url-pattern>
</servlet-mapping>
<welcome-file-list>
  <welcome-file>index.html</welcome-file>
</welcome-file-list>
</web-app>
```

Listing A.10: index.xml

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<!-- The HTML 4.01 Transitional DOCTYPE declaration-->
<!-- above set at the top of the file will set      -->
<!-- the browser's rendering engine into           -->
<!-- "Quirks Mode". Replacing this declaration     -->
<!-- with a "Standards Mode" doctype is supported, -->
<!-- but may lead to some differences in layout.   -->

<html>
  <head>
    <meta http-equiv="content-type" content="text/html; charset
      =UTF-8">
    <title>TestSingleTable</title>
  </head>

  <body>
    <h1>TestSingleTable</h1>

    <table>
      <tr>
        <td colspan="2" style="font-weight:bold;">Available
          Servlets:</td>
      </tr>
      <tr>
        <td><a href="main">Main</a></td>
      </tr>
    </table>
  </body>
</html>
```

A.2 test-crud-one-to-many

Listing A.11: Progetto.java

```
package entities;

import java.io.Serializable;
import java.lang.Long;
import java.lang.String;
import java.util.ArrayList;
import java.util.List;

import javax.persistence.*;

@Entity
@Table(name = "progetti0tM")
public class Progetto implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String descrizione;
    @OneToMany(cascade=CascadeType.ALL, mappedBy="progetto",
        orphanRemoval=true)
    private List<Attivita> attivita;
    private static final long serialVersionUID = 1L;
    public Progetto() {
        super();
        attivita=new ArrayList<Attivita>();
    } //costr

    public Long getId() {
        return this.id;
    } //getId
    public void setId(Long id) {
        this.id = id;
    } //setId

    public String getDescrizione() {
        return this.descrizione;
    } //getDescrizione
    public void setDescrizione(String descrizione) {
        this.descrizione = descrizione;
    } //setDescrizione

    public List<Attivita> getAttivita() {
        return attivita;
    } //getAttivita
    public void setAttivita(List<Attivita> attivita) {
        this.attivita = attivita;
    } //setAttivita

    @Override
    public String toString() {
```

```

        String s="Progetto [id=" + id + ", description=" +
            descrizione + ", hash=" + this.hashCode()+ "]" ;
        if(attivita!=null)
            for(Attivita a:attivita)
                s+="\n\t\t"+a;
        return s;
    }//toString
} //Progetto

```

Listing A.12: Attivita.java

```

package entities;

import java.io.Serializable;

import org.datanucleus.jpa.annotations.Extension;
import org.eclipse.persistence.annotations.Convert;
import org.eclipse.persistence.annotations.TypeConverter;
import org.hibernate.annotations.Type;
import org.hibernate.annotations.TypeDef;
import dao.HibernateConverter;

import javax.persistence.*;

@Entity
@TypeConverter(name="longToStringEL",dataType=Long.class,
    objectType=String.class) //convertitore string-long per
    EclipseLink
@TypeDef(name = "longToStringHI", typeClass =
    HibernateConverter.class)
@Table(name = "attivitaOtM")
public class Attivita implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Extension(vendorName="datanucleus", key="gae.encoded-pk",
        value="true") //richiesto dalla gestione delle key in
        AppEngine

    @Convert("longToStringEL")
    @Type(type="longToStringHI")
    private String id;
    private String descrizione;
    @ManyToOne
    private Progetto progetto;
    private static final long serialVersionUID = 1L;

    public Attivita() {
        super();
    } //costr

    public Attivita(Progetto p) {
        this();
        this.setProgetto(p);
    } //costr

```

```

public String getId() {
    return this.id;
} //getId
public void setId(String id) {
    this.id = id;
} //setId

public String getDescrizione() {
    return this.descrizione;
} //getDescrizione
public void setDescrizione(String descrizione) {
    this.descrizione = descrizione;
} //setDescrizione

public Progetto getProgetto() {
    return progetto;
} //getProgetto
public void setProgetto(Progetto progetto) {
    this.progetto = progetto;
} //setProgetto

@Override
public String toString(){
    return "Attivita [id=" + id + ", description=" +
        descrizione + ", progetto="+progetto.getId()+", hash="
        + this.hashCode()+ "]" ;
} //toString
} //Progetto

```

Listing A.13: EntityGenerator.java

```

package main;

import java.io.PrintWriter;
import java.util.ArrayList;

import dao.CrudProviderImpl;
import entities.Attivita;
import entities.Progetto;

public class EntityGenerator {
    private PrintWriter out;
    private CrudProviderImpl.Provider provider;

    public EntityGenerator(PrintWriter out, CrudProviderImpl.
        Provider provider){
        this.out=out;
        this.provider=provider;
    } //costr.

    public void doTests(){

```

```

        CrudProviderImpl<Progetto,Long> crud=new CrudProviderImpl<
            Progetto,Long>(provider);
        out.println("Inizio Test one to many: "+provider.toString()
            );
        Long id=create(crud,out);
        retrieve(crud,id);
        update(crud,id);
        //destroy(crud,id);
        out.println("Fine Test");
        crud.close();
    }//doTests

    private Long create (CrudProviderImpl<Progetto,Long> crud,
        PrintWriter out){
        out.print("C\t");
        Progetto p=new Progetto();
        p.setDescrizione("Nuovo");
        Attivita a=new Attivita(p);
        a.setDescrizione("Nuova");
        p.getAttivita().add(a);
        crud.create(p);
        out.println(p);
        return p.getId();
    }//create
    private void retrieve (CrudProviderImpl<Progetto,Long> crud,
        Long id){
        out.print("R\t");
        Progetto p=crud.retrieve(Progetto.class,id);
        out.println(p);
    }//retrieve

    private void update (CrudProviderImpl<Progetto,Long> crud,
        Long id){
        out.print("U\t");
        Progetto p=crud.retrieve(Progetto.class,id);
        p.setDescrizione("Modificato");
        crud.update(p);
        out.println(p);
    }//update

    private void destroy (CrudProviderImpl<Progetto,Long> crud,
        Long id){
        out.print("D\t");
        Progetto p=crud.retrieve(Progetto.class,id);
        if(p!=null){
            crud.destroy(p);
            out.println(p);
        }else
            out.println("Non trovo l'oggetto da eliminare");
    }//destroy
} //TestSingleTable

```

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/
  persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
  instance" xsi:schemaLocation="http://java.sun.com/xml/ns/
  persistence http://java.sun.com/xml/ns/persistence/
  persistence_2_0.xsd">
  <persistence-unit name="ECLIPSE_LINK" transaction-type="
    RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</
    provider>
    <class>entities.Progetto</class>
    <class>entities.Attivita</class>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:
        postgresql:progetti"/>
      <property name="javax.persistence.jdbc.user" value="
        postgres"/>
      <property name="javax.persistence.jdbc.password" value="
        torrenera"/>
      <property name="javax.persistence.jdbc.driver" value="org
        .postgresql.Driver"/>
    </properties>
  </persistence-unit>
  <persistence-unit name="HIBERNATE" transaction-type="
    RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>entities.Progetto</class>
    <class>entities.Attivita</class>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:
        postgresql:progetti"/>
      <property name="javax.persistence.jdbc.user" value="
        postgres"/>
      <property name="javax.persistence.jdbc.password" value="
        torrenera"/>
      <property name="javax.persistence.jdbc.driver" value="org
        .postgresql.Driver"/>
    </properties>
  </persistence-unit>
  <persistence-unit name="APP_ENGINE">
    <provider>org.datanucleus.store.appengine.jpa.
      DatastorePersistenceProvider</provider>
    <class>entities.Progetto</class>
    <class>entities.Attivita</class>
    <properties>
      <property name="datanucleus.NontransactionalRead" value="
        true"/>
      <property name="datanucleus.NontransactionalWrite" value
        ="true"/>
      <property name="datanucleus.ConnectionURL" value="
        appengine"/>
    </properties>
  </persistence-unit>

```

</persistence>

Listing A.15: HibernateConverter.java

```
package dao;

import java.io.Serializable;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Types;

import org.hibernate.HibernateException;
import org.hibernate.usertype.UserType;

public class HibernateConverter implements UserType{

    @Override
    public Object assemble(Serializable arg0, Object arg1)
        throws HibernateException {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public Object deepCopy(Object arg0) throws HibernateException
    {
        return arg0;
    }

    @Override
    public Serializable disassemble(Object arg0) throws
        HibernateException {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public boolean equals(Object x, Object y) throws
        HibernateException {
        if (x == y)
            return true;
        else if (x == null || y == null)
            return false;
        else
            return x.equals(y);
    }

    @Override
    public int hashCode(Object arg0) throws HibernateException {
        // TODO Auto-generated method stub
        return 0;
    }
}
```

```
@Override
public boolean isMutable() {
    // TODO Auto-generated method stub
    return false;
}

@Override
public Object nullSafeGet(ResultSet arg0, String[] arg1,
    Object arg2)
    throws HibernateException, SQLException {
    return Long.toString(arg0.getLong(arg1[0]));
}

@Override
public void nullSafeSet(PreparedStatement arg0, Object arg1,
    int arg2)
    throws HibernateException, SQLException {
    arg0.setLong(arg2, Long.parseLong((String)arg1));
}

@Override
public Object replace(Object arg0, Object arg1, Object arg2)
    throws HibernateException {
    // TODO Auto-generated method stub
    return null;
}

@Override
public Class<String> returnedClass() {
    return String.class;
}

@Override
public int[] sqlTypes() {
    int[] types={Types.BIGINT};
    return types;
}

}
```

A.3 test-crud-many-to-many

Listing A.16: Progetto.java

```
package entities;

import java.io.Serializable;
import java.lang.Long;
import java.lang.String;
import java.util.ArrayList;
import java.util.List;

import javax.persistence.*;

@Entity
@Table(name = "progettiMtM")
public class Progetto implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String descrizione;
    @ManyToMany(cascade=CascadeType.ALL, mappedBy="progetti")
    private List<Addetto> addetti;
    private static final long serialVersionUID = 1L;
    public Progetto() {
        super();
        addetti=new ArrayList<Addetto>();
    } //costr

    public Long getId() {
        return this.id;
    } //getId
    public void setId(Long id) {
        this.id = id;
    } //setId

    public String getDescrizione() {
        return this.descrizione;
    } //getDescrizione
    public void setDescrizione(String descrizione) {
        this.descrizione = descrizione;
    } //setDescrizione

    public List<Addetto> getAddetti() {
        return addetti;
    } //getAttivita
    public void setAddetti(List<Addetto> addetti) {
        this.addetti = addetti;
    } //setAttivita

    @Override
    public String toString() {
```

```

        String s="Progetto [id=" + id + ", description=" +
            descrizione + ", hash=" + this.hashCode()+ "]" ;
        if(addetti!=null)
            for(Addetto a:addetti)
                s+="\n\t\t"+a;
        return s;
    }//toString
}//Progetto

```

Listing A.17: Addetto.java

```

package entities;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

import javax.persistence.*;

@Entity
@Table(name = "addettiMtM")
public class Addetto implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String descrizione;
    @ManyToMany
    private List<Progetto> progetti;
    private static final long serialVersionUID = 1L;

    public Addetto() {
        super();
        progetti=new ArrayList<Progetto>();
    }//costr

    public Addetto(Progetto p) {
        this();
        this.progetti.add(p);
    }//costr

    public Long getId() {
        return this.id;
    }//getI
    public void setId(Long id) {
        this.id = id;
    } //setId

    public String getDescrizione() {
        return this.descrizione;
    }//getDescrizione
    public void setDescrizione(String descrizione) {
        this.descrizione = descrizione;
    }//setDescrizione

```

```

@Override
public String toString(){
    return "Addetto [id=" + id + ", description=" + descrizione
        + ", progetti="+progetti.size()+", hash=" + this.
            hashCode()+ "]";
} //toString
} //Progetto

```

Listing A.18: EntityGenerator.java

```

package main;

import java.io.PrintWriter;
import dao.CrudProviderImpl;
import entities.Addetto;
import entities.Progetto;

public class EntityGenerator {
    private PrintWriter out;
    private CrudProviderImpl.Provider provider;

    public EntityGenerator(PrintWriter out, CrudProviderImpl.
        Provider provider){
        this.out=out;
        this.provider=provider;
    } //costr.

    public void doTests(){
        CrudProviderImpl<Progetto, Long> crud=new CrudProviderImpl<
            Progetto, Long>(provider);
        out.println("Inizio Test many to many: "+provider.toString
            ());
        Long id=create(crud, out);
        //retrieve(crud, id);
        //update(crud, id);
        //destroy(crud, id);
        out.println("Fine Test");
        crud.close();
    } //doTests

    private Long create (CrudProviderImpl<Progetto, Long> crud,
        PrintWriter out){
        out.print("C\t");
        Progetto p=new Progetto();
        p.setDescrizione("Nuovo progetto");
        Addetto a=new Addetto(p);
        a.setDescrizione("Nuovo addetto");
        p.getAddetti().add(a);
        crud.create(p);
        out.println(p);
        return p.getId();
    }

```

```

    }//create
    private void retrieve (CrudProviderImpl<Progetto,Long> crud,
        Long id){
        out.print("R\t");
        Progetto p=crud.retrieve(Progetto.class,id);
        out.println(p);
    }//retrieve

    private void update (CrudProviderImpl<Progetto,Long> crud,
        Long id){
        out.print("U\t");
        Progetto p=crud.retrieve(Progetto.class,id);
        p.setDescrizione("Modificato");
        crud.update(p);
        out.println(p);
    }//update

    private void destroy (CrudProviderImpl<Progetto,Long> crud,
        Long id){
        out.print("D\t");
        Progetto p=crud.retrieve(Progetto.class,id);
        if(p!=null){
            crud.destroy(p);
            out.println(p);
        }else
            out.println("Non trovo l'oggetto da eliminare");
    }//destroy
} //TestSingleTable

```

Listing A.19: persistence.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/
  persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
  instance" xsi:schemaLocation="http://java.sun.com/xml/ns/
  persistence http://java.sun.com/xml/ns/persistence/
  persistence_2_0.xsd">
  <persistence-unit name="ECLIPSE_LINK" transaction-type="
    RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</
      provider>
    <class>entities.Progetto</class>
    <class>entities.Addetto</class>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:
        postgresql:progetti"/>
      <property name="javax.persistence.jdbc.user" value="
        postgres"/>
      <property name="javax.persistence.jdbc.password" value="
        torrenera"/>
      <property name="javax.persistence.jdbc.driver" value="org
        .postgresql.Driver"/>
    </properties>
  </persistence-unit>
</persistence>

```

```
</persistence-unit>
<persistence-unit name="HIBERNATE" transaction-type="
    RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>entities.Progetto</class>
    <class>entities.Addetto</class>
    <properties>
        <property name="javax.persistence.jdbc.url" value="jdbc:
            postgresql:progetti"/>
        <property name="javax.persistence.jdbc.user" value="
            postgres"/>
        <property name="javax.persistence.jdbc.password" value="
            torrenera"/>
        <property name="javax.persistence.jdbc.driver" value="org
            .postgresql.Driver"/>
    </properties>
</persistence-unit>
<persistence-unit name="APP_ENGINE">
    <provider>org.datanucleus.store.appengine.jpa.
        DatastorePersistenceProvider</provider>
    <class>entities.Progetto</class>
    <class>entities.Addetto</class>
    <properties>
        <property name="datanucleus.NontransactionalRead" value="
            true"/>
        <property name="datanucleus.NontransactionalWrite" value
            ="true"/>
        <property name="datanucleus.ConnectionURL" value="
            appengine"/>
    </properties>
</persistence-unit>

</persistence>
```

A.4 test-crud-jdbc

Listing A.20: Main.java

```
import java.io.PrintWriter;
import main.QueryTester;

public class Main {
    public static void main(String[] args) {
        jdbcBridge.Connection.setProvider("HIBERNATE");
        new QueryTester(new PrintWriter(System.out,true)).doTests();
    }
} //main
} //Main
```

Listing A.21: Connection.java

```
package jdbcBridge;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class Connection {
    private static EntityManagerFactory emf;
    private static EntityManager em;
    private static String provider;
    public Connection(){
        emf = Persistence.createEntityManagerFactory(provider);
        em = emf.createEntityManager();
    } //costr
    public Statement createStatement(){
        return new Statement(em);
    }

    public static void setProvider(String provider) {
        Connection.provider = provider;
    }
    public static String getProvider() {
        return provider;
    }
}
```

Listing A.22: DriverManager.java

```
package jdbcBridge;

public class DriverManager {
    public static Connection getConnection(String connection,
        String user,String password){
        return new Connection();
    }
}
```

```
}  
  
}
```

Listing A.23: ResultSet.java

```
package jdbcBridge;  
  
import java.util.List;  
  
public class ResultSet {  
    @SuppressWarnings("rawtypes")  
    private List l;  
    private Object currentObject;  
    private int idxObject;  
    public ResultSet(@SuppressWarnings("rawtypes") List l){  
        this.l=l;  
        idxObject=0;  
    } //costr  
  
    public boolean next(){  
        if(idxObject<l.size()){  
            currentObject=l.get(idxObject++);  
            return true;  
        }  
        else return false;  
    } //next  
  
    public String getString(int index) throws SQLException{  
        try{  
            try{  
                Object[] row=(Object[])currentObject;  
                return row[index-1].toString();  
            }catch(Exception e){  
            } //catch  
            if(index>1)  
                throw new SQLException(new Exception("Index out of  
                    bound in getString"));  
            return currentObject.toString();  
        }catch(Exception e){  
            throw new SQLException(e);  
        }  
    }  
}
```

Listing A.24: SQLException.java

```
package jdbcBridge;  
  
public class SQLException extends Exception{  
    private static final long serialVersionUID = 1L;
```

```
    public SQLException(Exception e){
        super(e);
    }
}
```

Listing A.25: Statement.java

```
package jdbcBridge;

import javax.persistence.EntityManager;
import javax.persistence.Query;

public class Statement {
    private EntityManager em;
    public Statement(EntityManager em){
        this.em=em;
    } //costr
    public ResultSet executeQuery(String query){
        Query q = em.createQuery(query);
        return new ResultSet(q.getResultList());
    }
    public boolean execute(String query){
        em.createQuery(query);
        return false;
    }
}
```

Listing A.26: MainServlet.java

```
package main;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.http.*;

@SuppressWarnings("serial")
public class MainServlet extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse
        resp) throws IOException {
        resp.setContentType("text/plain");
        jdbcBridge.Connection.setProvider("APP_ENGINE");
        new QueryTester(new PrintWriter(resp.getWriter(),true)).
            doTests();

    } //doGet
} //MainServlet
```

Listing A.27: QueryTester.java

```
package main;
```



```

import java.io.PrintWriter;
import java.util.ArrayList;

public class QueryTester {
    private ArrayList<String> campioni;
    private PrintWriter out;
    public QueryTester(PrintWriter out){
        initCampioni();
        this.out=out;
    } //costr

    private void initCampioni(){
        campioni=new ArrayList<String>();
        campioni.add("SELECT * FROM Progetto p");
        campioni.add("SELECT p FROM Progetto p");
        campioni.add("SELECT descrizione FROM Progetto");
        campioni.add("SELECT p.descrizione FROM Progetto p");
        campioni.add("SELECT a.descrizione FROM Progetto p INNER
            JOIN Attivita a ON p.id=a.progetto_id");
        campioni.add("SELECT a.id FROM Progetto p INNER JOIN p.
            attivita a");
        campioni.add("SELECT p.id FROM Progetto p INNER JOIN p.
            attivita a");
        campioni.add("SELECT MAX(p.id) FROM Progetto p");
        campioni.add("SELECT p.id FROM Progetto p ORDER BY id DESC
            ");
        campioni.add("INSERT INTO Progetto VALUES (300,'Temp',NULL)
            ");
        campioni.add("UPDATE Progetto p SET descrizione='updated'
            WHERE id=300");
        campioni.add("DELETE FROM Progetto WHERE id=300");
    }

    public void doTests(){
        int i=0;
        for(String t:campioni){
            out.println(++i+" "+t);
            if(!jdbcBridge.Connection.getProvider().equals("
                APP_ENGINE")){
                //sezione SQL su JDBC
                try{
                    java.sql.Connection con = java.sql.DriverManager.
                        getConnection("jdbc:postgresql:progetti", "
                            postgres","torrenera");
                    if(t.startsWith("SELECT")){
                        java.sql.ResultSet rs = con.createStatement().
                            executeQuery(t);
                        out.println("\tSUCCESS");
                        while (rs.next())
                            out.println("\t\t"+rs.getString(1));
                    }else{
                        con.createStatement().execute(t);
                        out.println("\tSUCCESS");
                    }
                }
            }
        }
    }
}

```

```
        }catch(Exception e){
            out.println("\tFAIL: "+e.getMessage());
        }//catch
    }
    //SEZIONE JPQL SU BRIDGE
    try{
        jdbcBridge.Connection con = jdbcBridge.DriverManager.
            getConnection("jdbc:postgresql:progetti", "postgres",
                "torrenera");
        if(t.startsWith("SELECT")){
            jdbcBridge.ResultSet rs = con.createStatement().
                executeQuery(t);
            out.println("\tSUCCESS");
            while (rs.next())
                out.println("\t\t"+rs.getString(1));
        }else{
            con.createStatement().execute(t);
            out.println("\tSUCCESS");
        }
    }catch(Exception e){
        out.println("\tFAIL: "+e.getMessage());
    }//catch
    out.println
        ("-----");
    ;
    }//for
    }//doTest
} //QueryTester
```

A.5 test-session

Listing A.28: SetAttributeServlet.java

```
package main;

import java.io.IOException;

import javax.servlet.http.*;

@SuppressWarnings("serial")
public class SetAttributeServlet extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse
        resp)
        throws IOException {
        resp.setContentType("text/plain");
        HttpSession session=req.getSession();
        resp.getWriter().println("Session id: "+session.getId());
        session.setAttribute("attributo1","valore1");
    }
}
```

Listing A.29: GetAttributeServlet.java

```
package main;

import java.io.IOException;

import javax.servlet.http.*;

@SuppressWarnings("serial")
public class GetAttributeServlet extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse
        resp)
        throws IOException {
        resp.setContentType("text/plain");
        HttpSession session=req.getSession();
        resp.getWriter().println("Session id: "+session.getId());
        resp.getWriter().println(session.getAttribute("attributo1")
        );
    }
}
```

Listing A.30: index.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<!-- The HTML 4.01 Transitional DOCTYPE declaration-->
<!-- above set at the top of the file will set -->
<!-- the browser's rendering engine into -->
<!-- "Quirks Mode". Replacing this declaration -->
<!-- with a "Standards Mode" doctype is supported, -->
```

```

<!-- but may lead to some differences in layout. -->

<html>
  <head>
    <meta http-equiv="content-type" content="text/html; charset
      =UTF-8">
    <title>Test su HttpSession</title>
  </head>

  <body>
    <h1>Test su HttpSession</h1>

    <table>
      <tr>
        <td colspan="2" style="font-weight:bold;">Available
          Servlets:</td>
      </tr>
      <tr>
        <td><a href="setAttribute">SetAttribute</a></td>
      </tr>
      <tr>
        <td><a href="getAttribute">GetAttribute</a></td>
      </tr>
    </table>
  </body>
</html>

```

Listing A.31: appengine-web.xml

```

<?xml version="1.0" encoding="utf-8"?>
<appengine-web-app xmlns="http://appengine.google.com/ns/1.0">
  <application>test-crud</application>
  <version>5</version>

  <!-- Configure java.util.logging -->
  <system-properties>
    <property name="java.util.logging.config.file" value="WEB-
      INF/logging.properties"/>
  </system-properties>
  <sessions-enabled>true</sessions-enabled>
</appengine-web-app>

```

Listing A.32: web.xml

```

<?xml version="1.0" encoding="utf-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version
    ="2.5">
  <servlet>

```

```
<servlet-name>SetAttribute</servlet-name>
<servlet-class>main.SetAttributeServlet</servlet-class>
</servlet>
<servlet>
  <servlet-name>GetAttribute</servlet-name>
  <servlet-class>main.GetAttributeServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>SetAttribute</servlet-name>
  <url-pattern>/setAttribute</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>GetAttribute</servlet-name>
  <url-pattern>/getAttribute</url-pattern>
</servlet-mapping>
<welcome-file-list>
  <welcome-file>index.html</welcome-file>
</welcome-file-list>
</web-app>
```

A.6 gae-time-datastore

Listing A.33: CrudProviderImpl.java

```
package dao;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;

public class CrudProviderImpl<T, K> implements CrudProvider<T,
    K> {
    private static EntityManagerFactory emf;
    private static EntityManager em;
    private static enum Provider{ECLIPSE_LINK,HIBERNATE,APP_ENGINE
    };

    public CrudProviderImpl(Provider p){
        emf = Persistence.createEntityManagerFactory(p.toString());
        em = emf.createEntityManager();
    } //costr.

    @Override
    public synchronized void create(T object) {
        EntityTransaction t=em.getTransaction();
        try{
            t.begin();
            em.persist(object);
            t.commit();
        }catch(Exception e){
            e.printStackTrace();
            t.rollback();
        } //catch
    } //create

    @Override
    public synchronized T retrieve(Class<T> c,K key) {
        return (T)em.find(c,key);
    } //retrieve

    @Override
    public synchronized T update(T object) {
        EntityTransaction t=em.getTransaction();
        try{
            t.begin();
            T aggiornato=em.merge(object);
            t.commit();
            return aggiornato;
        }catch(Exception e){
            e.printStackTrace();
            t.rollback();
            return null;
        }
    }
}
```

```

        } // catch
    } // update

    @Override
    public synchronized void destroy(T object) {
        EntityTransaction t = em.getTransaction();
        try {
            t.begin();
            em.remove(em.merge(object));
            t.commit();
        } catch (Exception e) {
            e.printStackTrace();
            t.rollback();
        } // catch
    } // destroy
    public synchronized void close() {
        em.close();
        emf.close();
    } // close
} // CrudProviderImpl

```

Listing A.34: Record.java

```

package entities;

import java.io.Serializable;
import java.lang.Long;
import java.lang.String;
import javax.persistence.*;

@Entity
@Table(name = "recordC")
public class Record implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String descrizione;
    private static final long serialVersionUID = 1L;

    public Record() {
        super();
    } // costr

    public Long getId() {
        return this.id;
    } // getId
    public void setId(Long id) {
        this.id = id;
    } // setId

    public String getDescrizione() {
        return this.descrizione;
    } // getDescrizione

```

```

    public void setDescription(String descrizione) {
        this.descrizione = descrizione;
    } // setDescription

    @Override
    public String toString() {
        return "Record [id=" + id + ", description=" + descrizione
            + ", hash=" + this.hashCode() + "]";
    } // toString
} // Progetto

```

Listing A.35: Test_time_datastoreServlet.java

```

package main;

import java.io.IOException;
import java.util.Random;

import javax.servlet.http.*;

import dao.CrudProviderImpl;
import entities.Record;

@SuppressWarnings("serial")
public class Test_time_datastoreServlet extends HttpServlet {
    private static final CrudProviderImpl<Record, Long> provider =
        new CrudProviderImpl<Record, Long>(CrudProviderImpl.
            Provider.APP_ENGINE);
    private static final int L=10;
    private static final long version=new Random().nextLong();
    public void doGet(HttpServletRequest req, HttpServletResponse
        resp)
        throws IOException {
        long start, total, time, max=Long.MIN_VALUE, min=Long.MAX_VALUE,
            sum=0;
        total=System.nanoTime();
        for(int i=0; i<L; i++){
            Record r=new Record();
            start=System.nanoTime();
            provider.create(r);
            time=System.nanoTime()-start;
            max=(time>max?time:max);
            min=(time<min?time:min);
            sum+=time;
        }
        resp.getOutputStream().println(version+"\t"+max/1e6+"\t"+
            min/1e6+"\t"+(sum/1e6/L)+"\t"+(System.nanoTime()-
            total)/1e6);
    }
}

```

A.7 local-time-datastore

Listing A.36: Main.java

```
public class Main {  
  
    public static void main(String[] args) {  
        new Launcher().doTest();  
    }  
}
```

Listing A.37: Launcher.java

```
import java.io.BufferedWriter;  
import java.io.FileWriter;  
import java.io.PrintWriter;  
import java.util.concurrent.Semaphore;  
  
public class Launcher {  
    public void doTest(){  
        int K=50; //500  
        int PARALLEL[]={1,2,4};//,8,16,32}; //,64,128,256,512};  
        for(int p:PARALLEL){  
            try {  
                PrintWriter out = new PrintWriter(new BufferedWriter(  
                    new FileWriter("/home/pier/tempi7-"+p+".csv",true))  
                    , true);  
                out.println("Istanza\tMax\tMin\tAvg\tServer\tClient");  
                Semaphore s=new Semaphore(p);  
                //K=(K<p?p:K);  
                for(int i=0;i<K;i++){  
                    SingleTest t=new SingleTest(i,s,out);  
                    t.start();  
                }//for  
                s.acquire(p);  
                out.close();  
            } catch (Exception e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

Listing A.38: SingleTest.java

```
import java.io.BufferedReader;  
import java.io.InputStreamReader;  
import java.io.PrintWriter;
```

```
import java.net.URL;
import java.net.URLConnection;
import java.util.concurrent.Semaphore;

public class SingleTest extends Thread{
    private int i;
    private Semaphore s;
    private PrintWriter out;
    public SingleTest(int i,Semaphore s,PrintWriter out){
        this.i=i;
        this.s=s;
        this.out=out;
    } //out
    public void run(){
        try {
            s.acquire();
        } catch (InterruptedException e1) {
            e1.printStackTrace();
        }
        System.out.println("Inizio"+i);
        URL url=null;
        URLConnection conn=null;
        long start,time;
        try{
            start=System.nanoTime();
            url = new URL("http://11.test-crud.appspot.com/
                test_time_datastore");
            //url = new URL("http://localhost:8888/
                test_time_datastore");
            //url = new URL("http://conte-00.appspot.com/
                test_time_datastore");
            conn = url.openConnection();
            BufferedReader rd = new BufferedReader(new
                InputStreamReader(conn.getInputStream()));
            String line;
            while ((line = rd.readLine()) != null){
                time=System.nanoTime()-start;
                out.println((line+"\t"+time/1e6).replace
                    (".",","));
                System.out.println(i+" "+line+"\t"+time/1e6);
            } //while
            rd.close();
        } catch (Exception e){
            e.printStackTrace();
        } finally{
            s.release();
        }
    } //run
}
```

B.1 local-time-datastore ($L=100$)

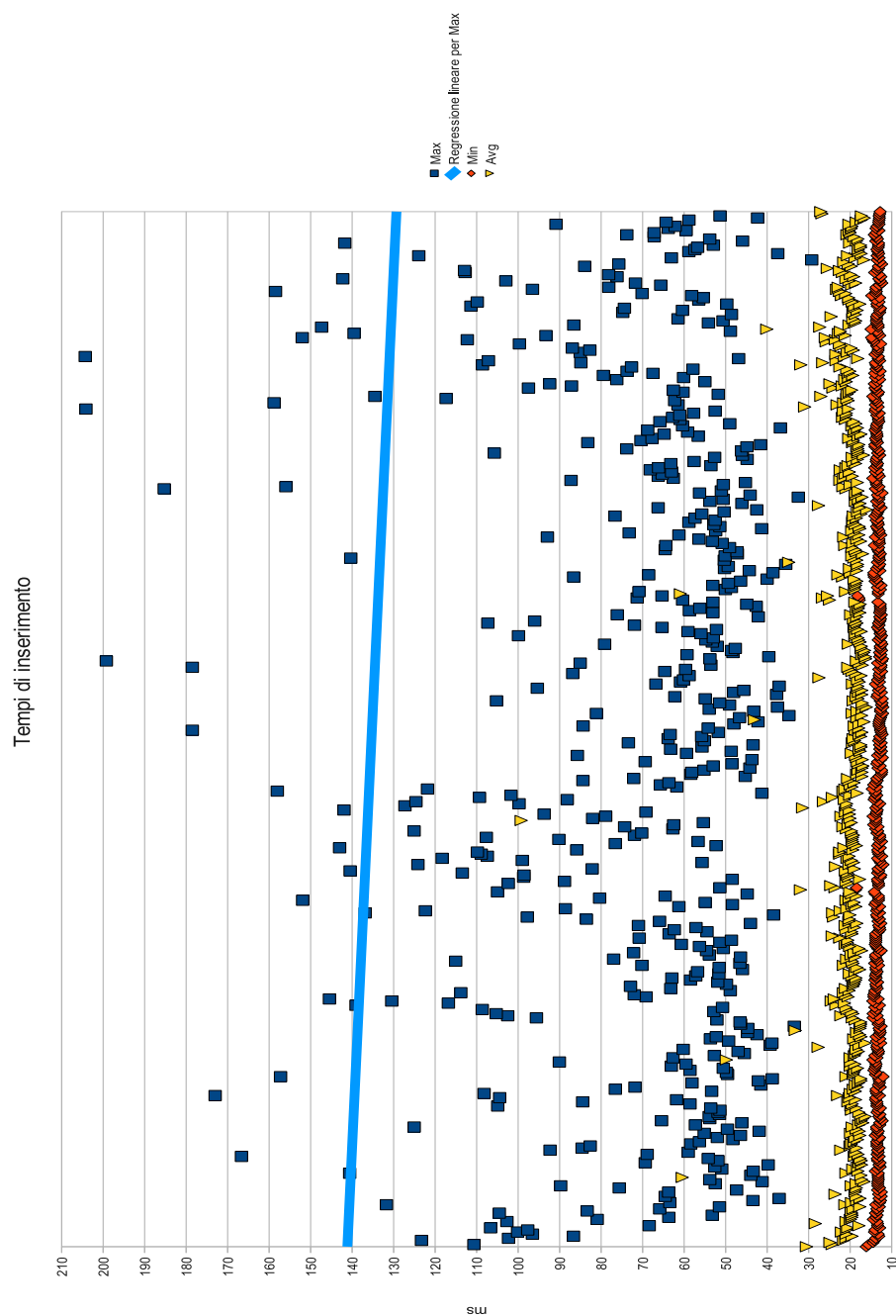
In questa sezione, verrà presentata una serie di grafici ritenuti significativi, a supporto dei test compiuti nella sezione 5.2.

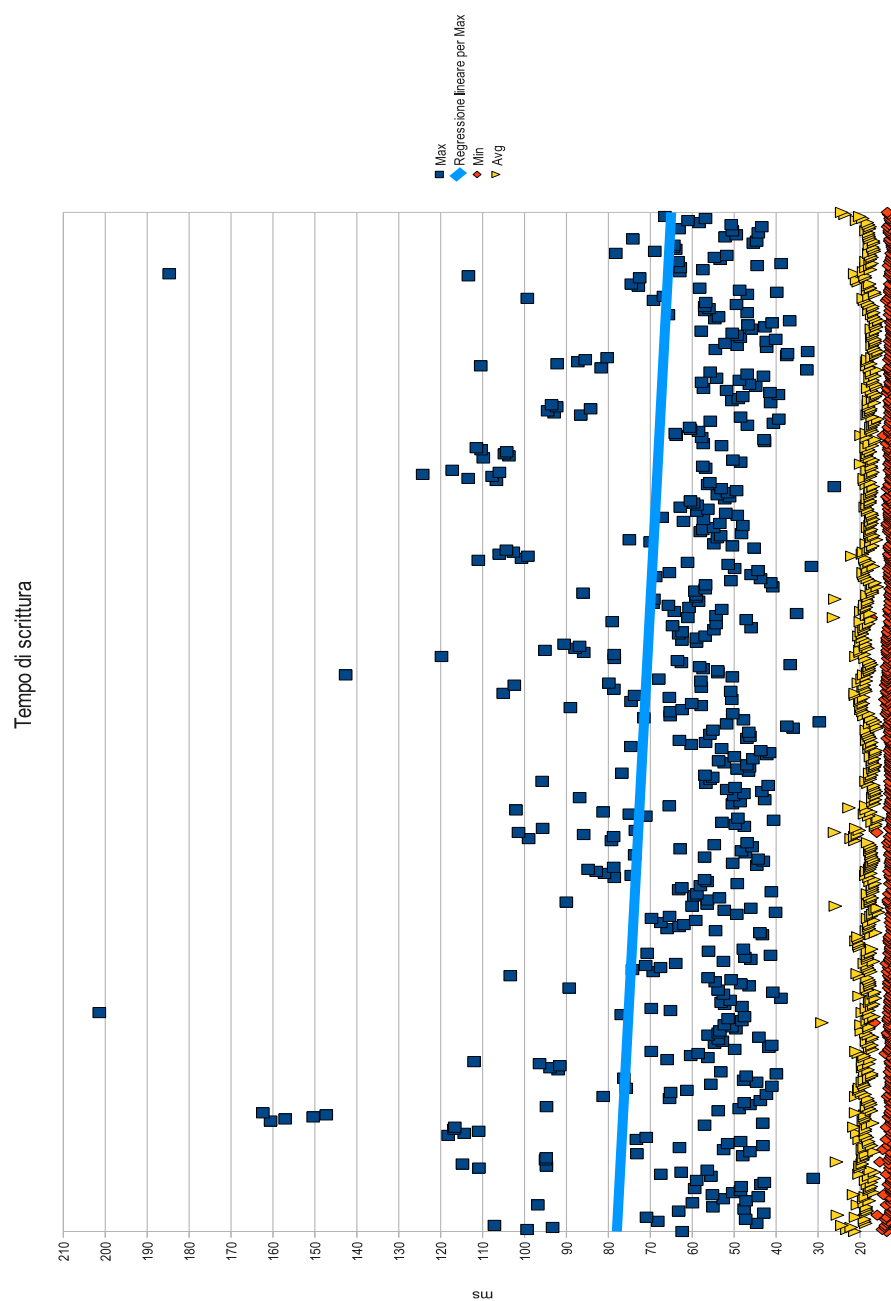
I grafici del tipo *Tempo di inserimento* evidenziano le latenze di inserimento della singola entità: i risultati sono generalmente ordinati in base all'ordine di arrivo delle risposte al client, tranne alcuni casi in cui vengono raggruppati per *id di istanza* al fine di evidenziare peculiarità relative al comportamento della singola istanza della servlet.

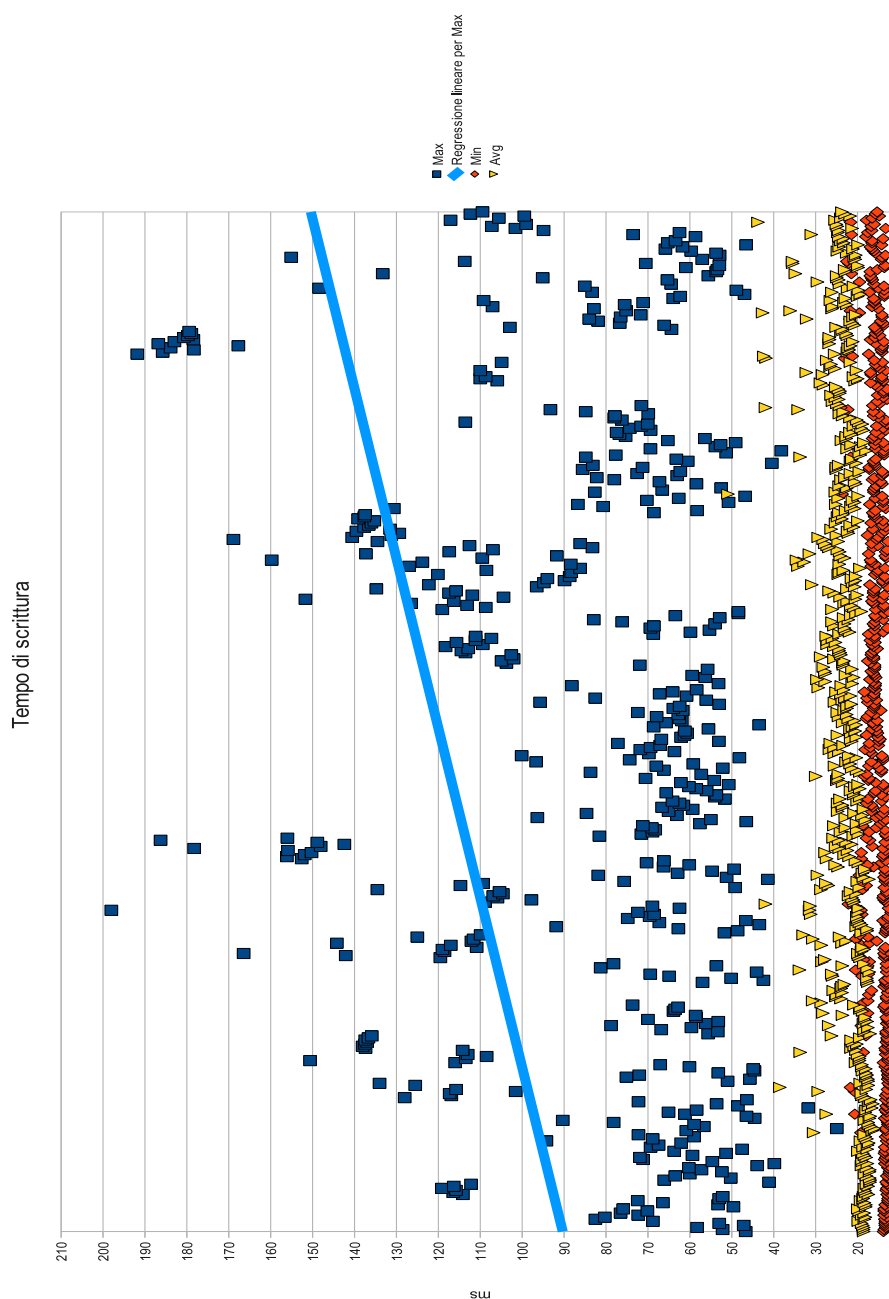
In questi grafici, vengono presentati i tempi massimi, minimi e medi (in millisecondi) rilevati all'interno della singola esecuzione della servlet, che prevede l'inserimento di L record nel datastore. In molti casi, è stata introdotta una linea di tendenza con andamento lineare sui valori massimi, al fine di avere un'informazione approssimativa dell'andamento di tali valori. Otteniamo quindi dei grafici che contengono K punti per ciascuna delle tre tipologie evidenziate, dai quali sono state ricavate delle osservazioni sulla distribuzione e sull'andamento dei valori misurati.

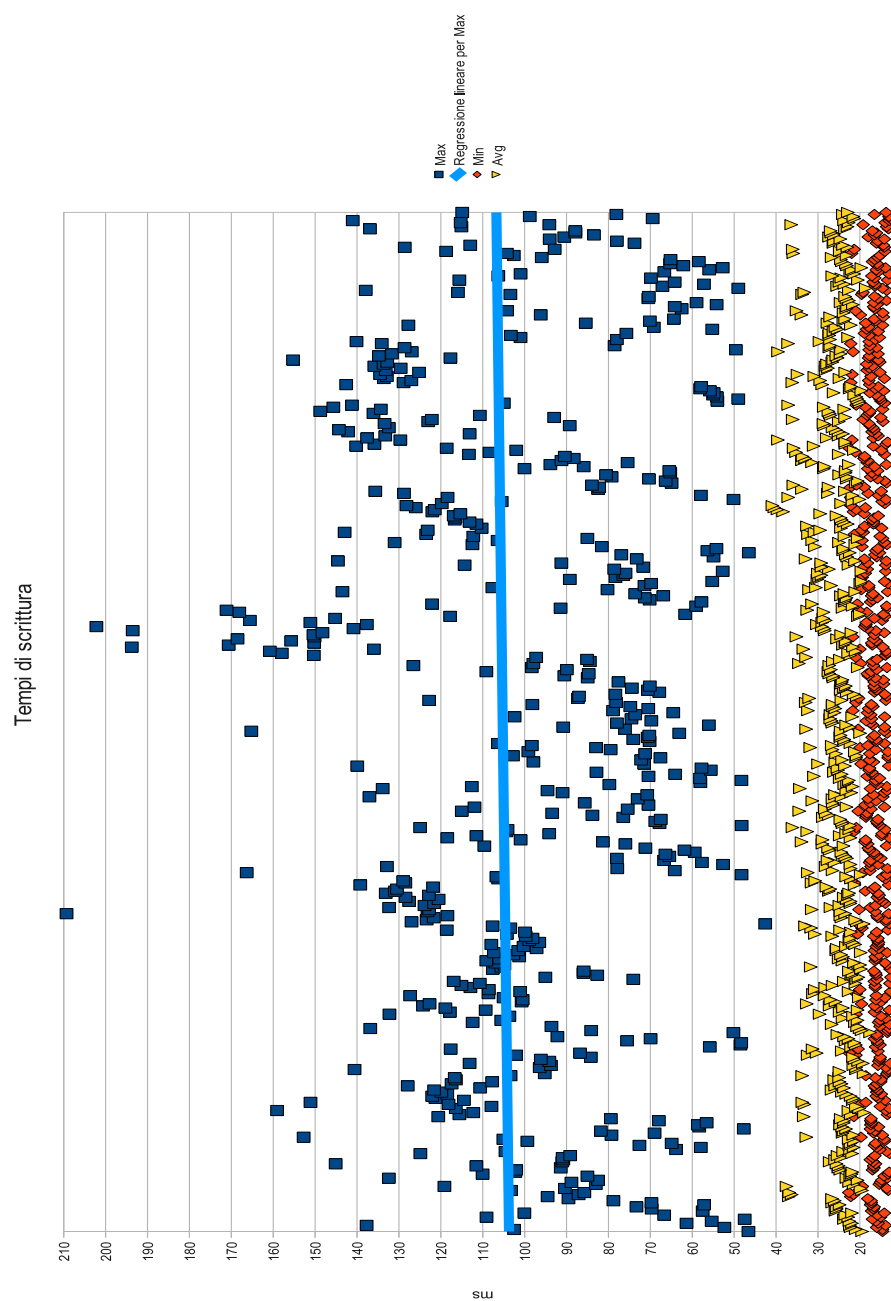
Viceversa, i grafici del *Tempo totale* mettono in relazione i tempi di risposta della servlet, misurato dal lato server e lato client, come illustrato nella sottosezione 5.2.6. Anche in questo caso i tempi sono espressi in millisecondi, e sono state introdotte delle linee di tendenza ad andamento lineare.

In determinate situazioni, la grande differenza dei valori rilevati tra client e server ha richiesto uno sdoppiamento dell'asse verticale per evitare un'eccessivo schiacciamento della rappresentazione dei valori dal lato server: in tali situazione, si ha l'asse principale a sinistra che contiene la scala dal lato server, abbinato ad un asse secondario a destra dedicato ai valori del lato client.

Figura B.1: Tempi di inserimento ($P=1$)

Figura B.2: Tempi di inserimento ($P=8$)

Figura B.3: Tempi di inserimento ($P=64$)

Figura B.4: Tempi di inserimento ($P=512$)

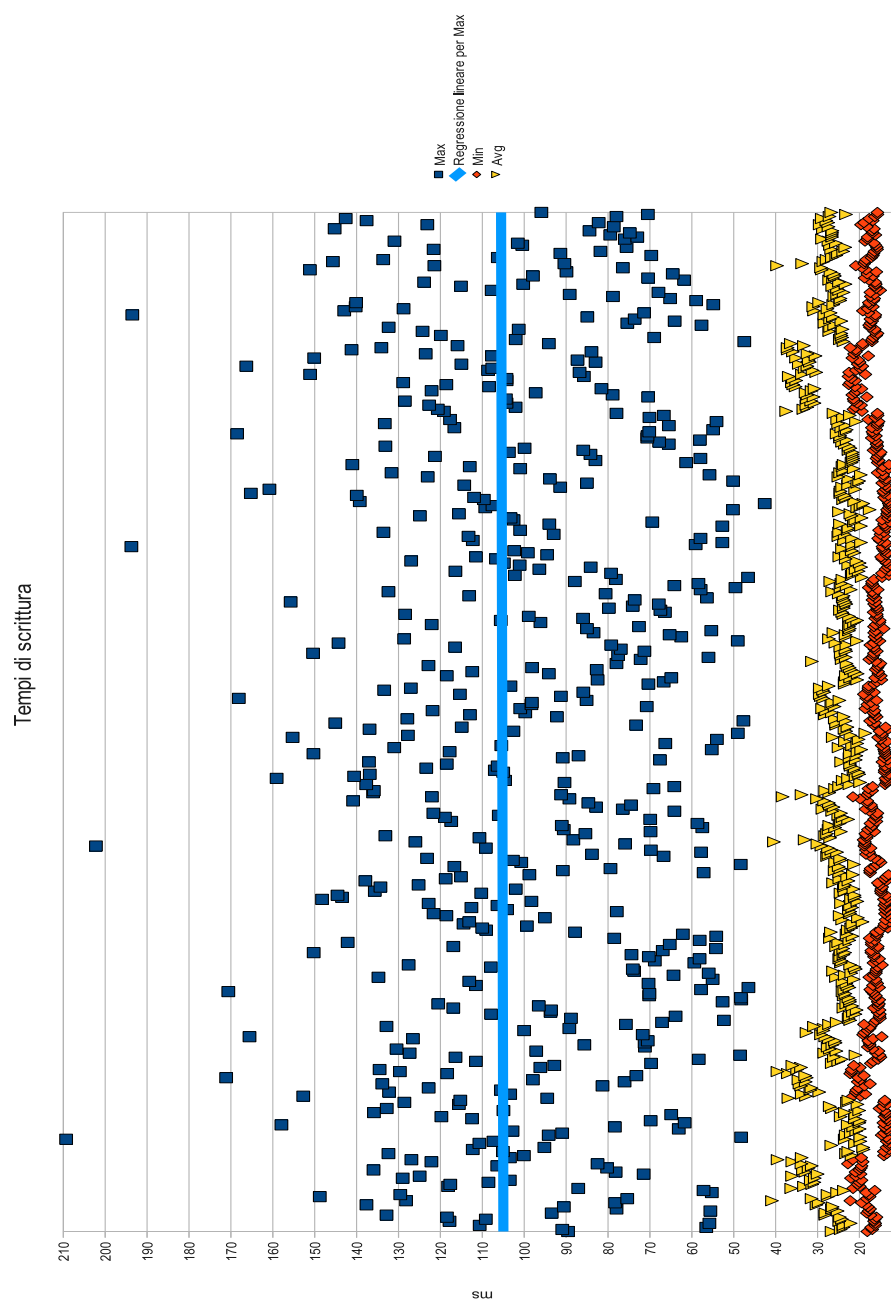
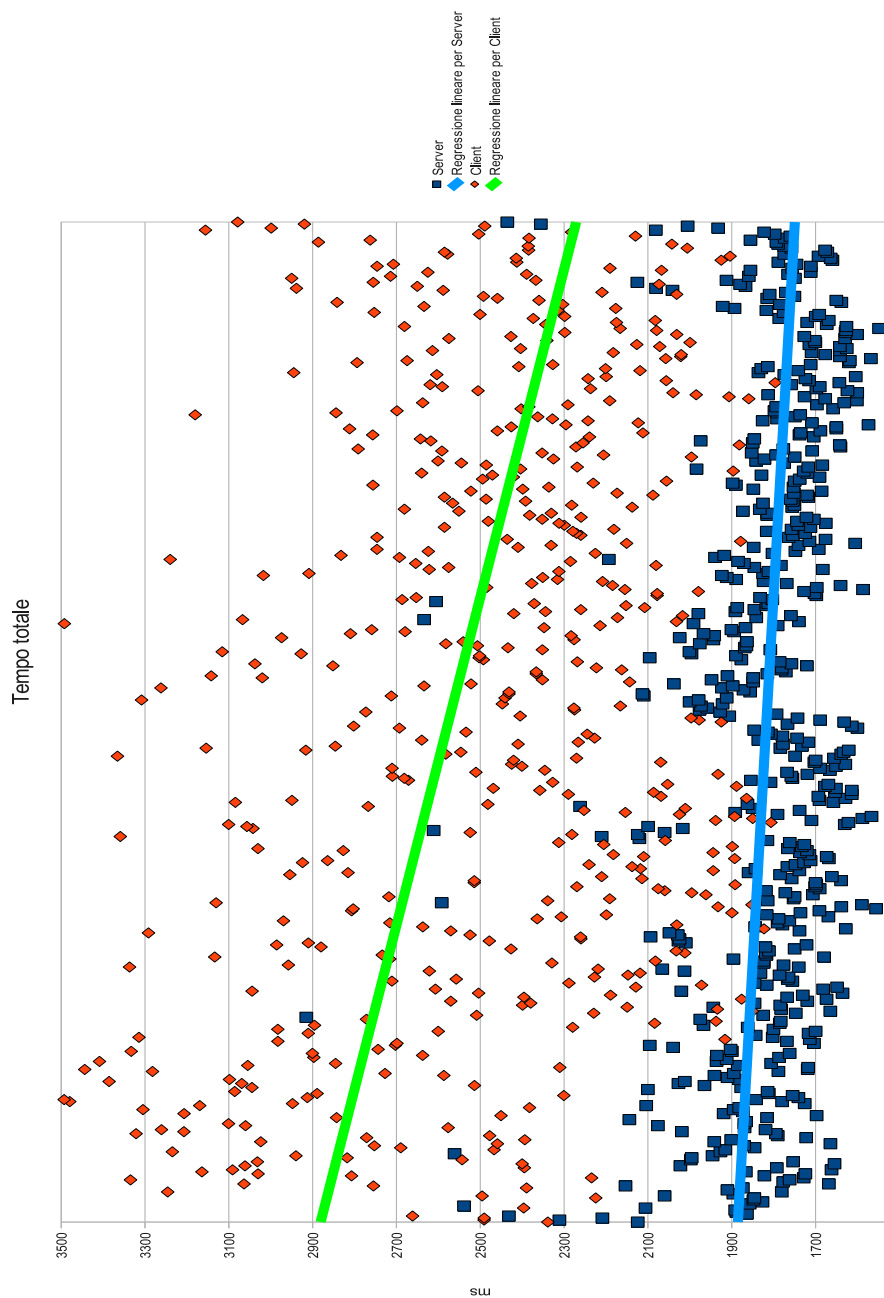
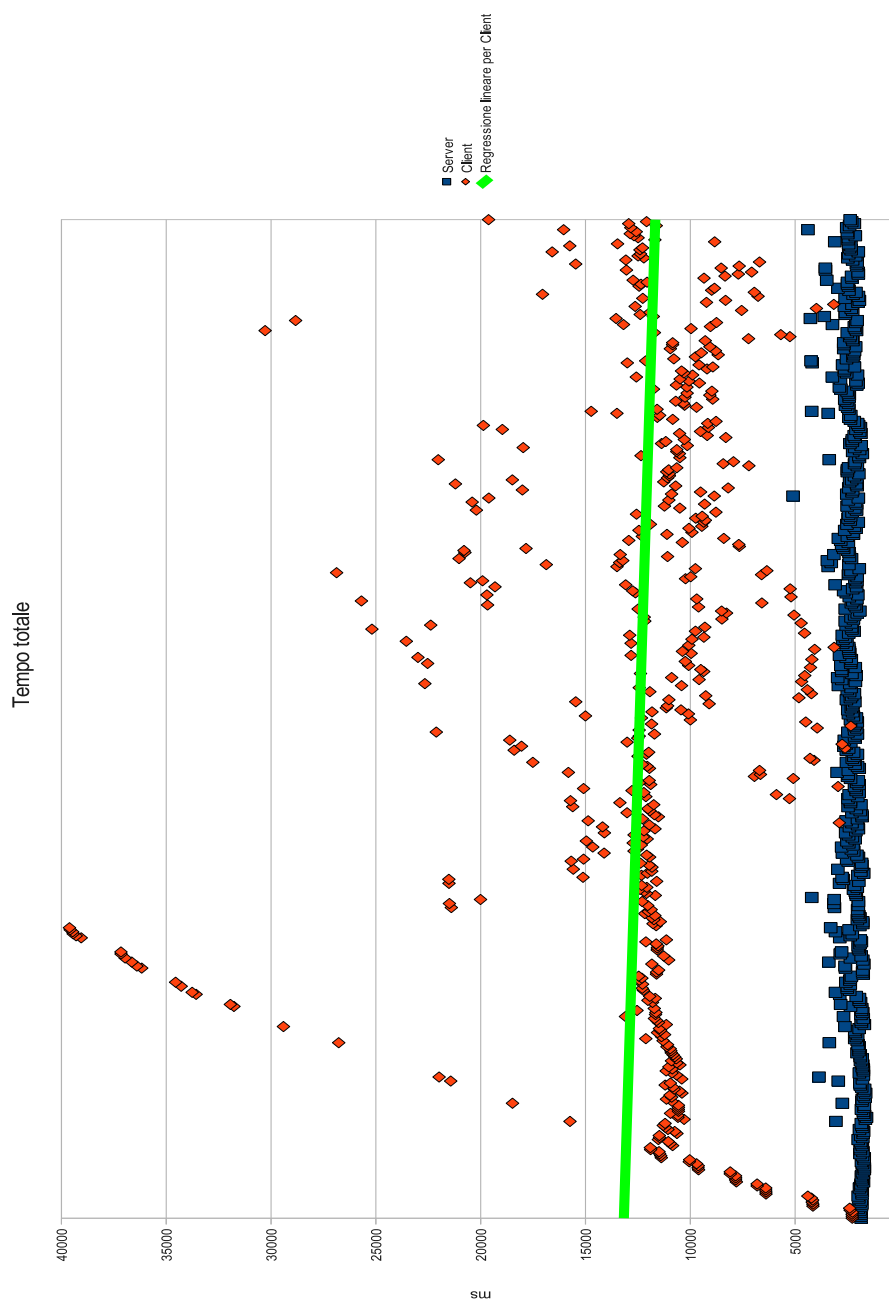
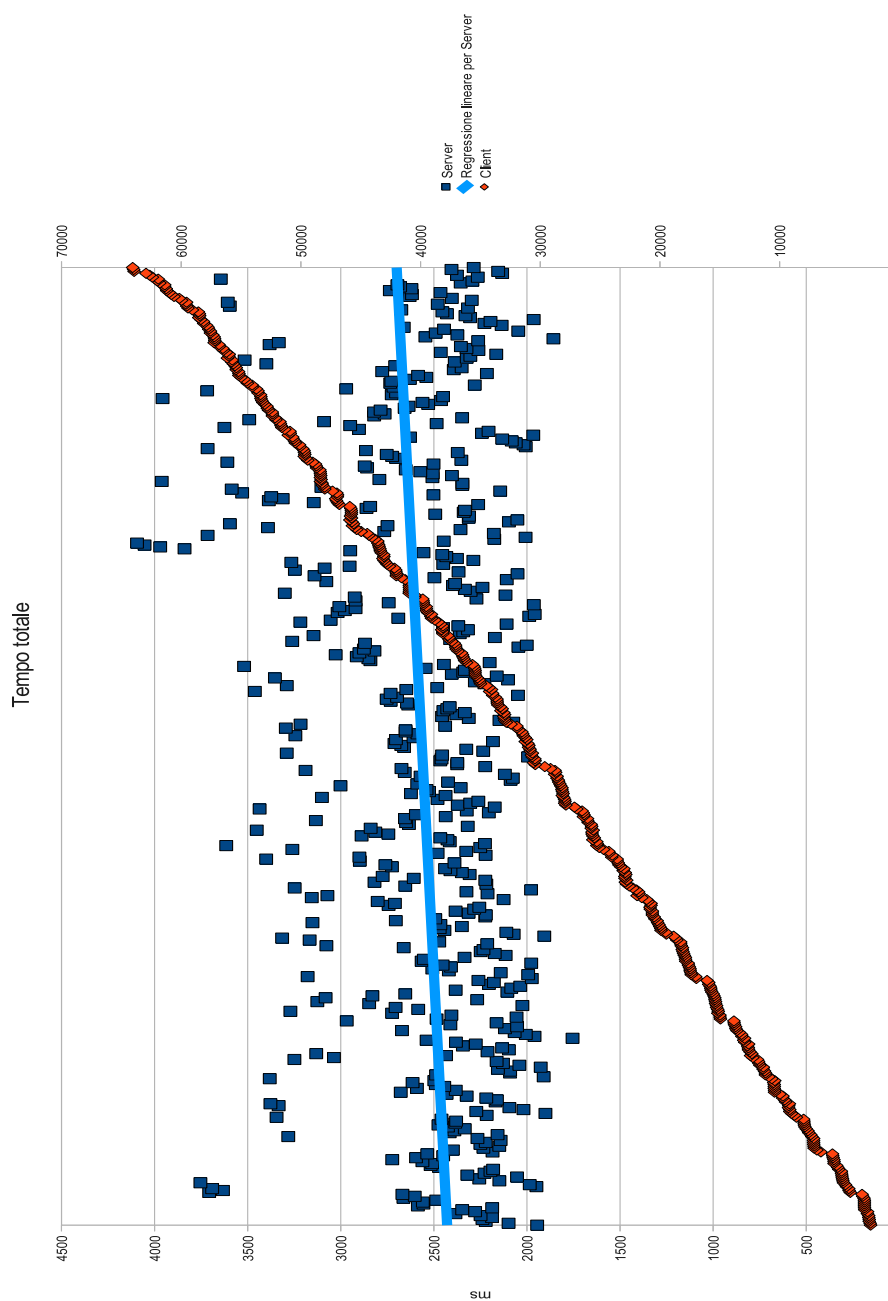


Figura B.5: Tempi di inserimento ($P=512$), per istanza

Figura B.6: Tempi totali di risposta ($P=1$)

Figura B.7: Tempi totali di risposta ($P=8$)

Figura B.8: Tempi totali di risposta ($P=64$)

Figura B.9: Tempi totali di risposta ($P=512$)

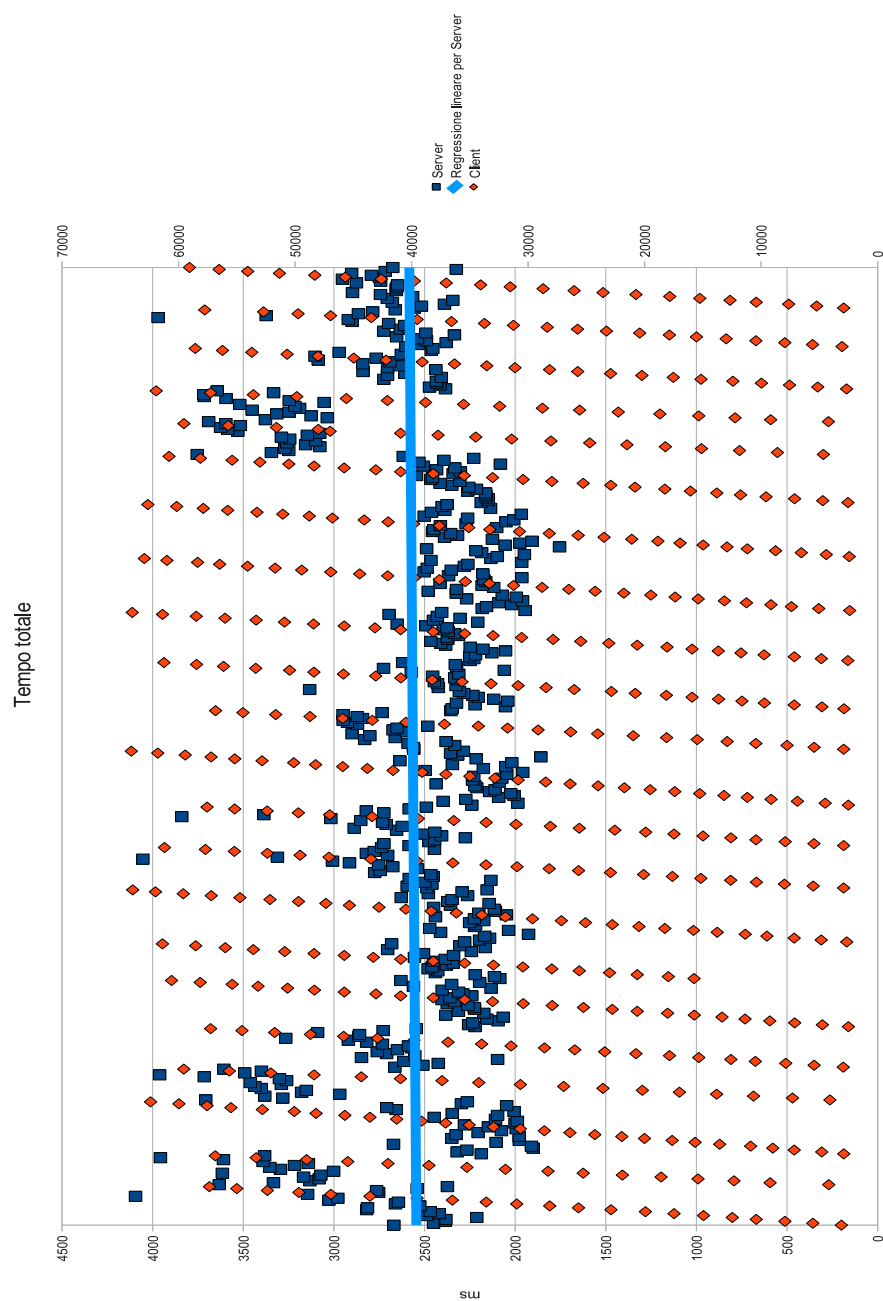


Figura B.10: Tempi totali di risposta ($P=512$), per istanza

B.2 local-time-datastore ($L=10$)

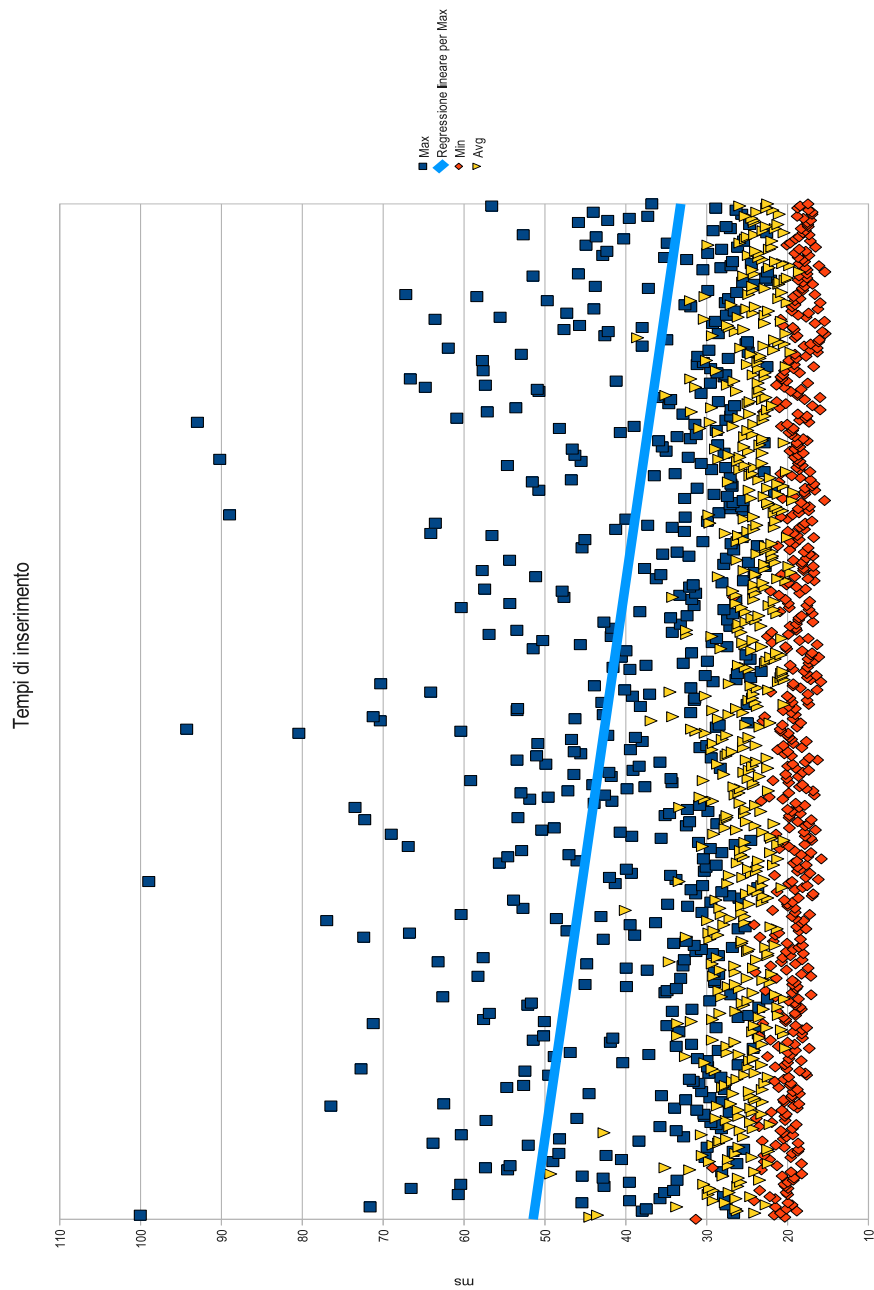
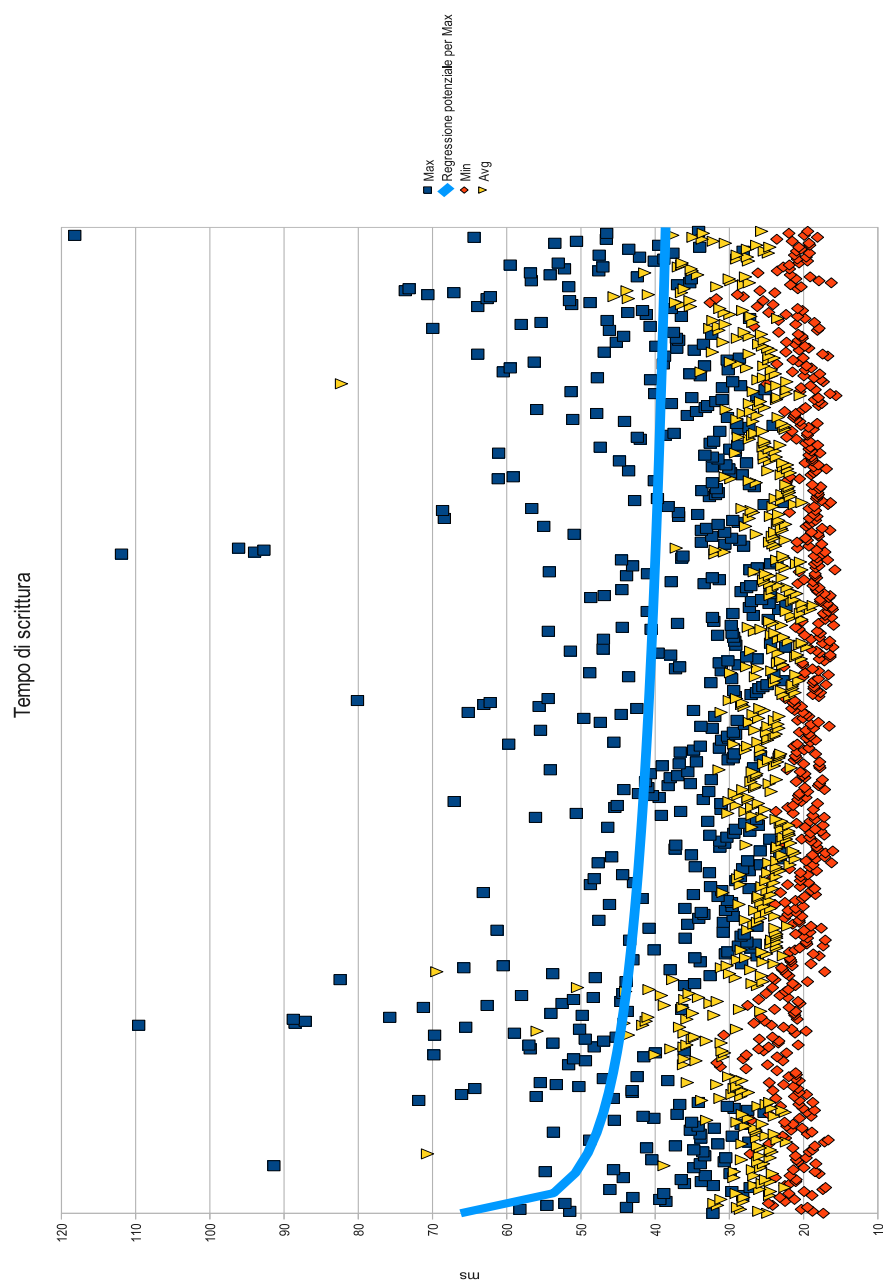
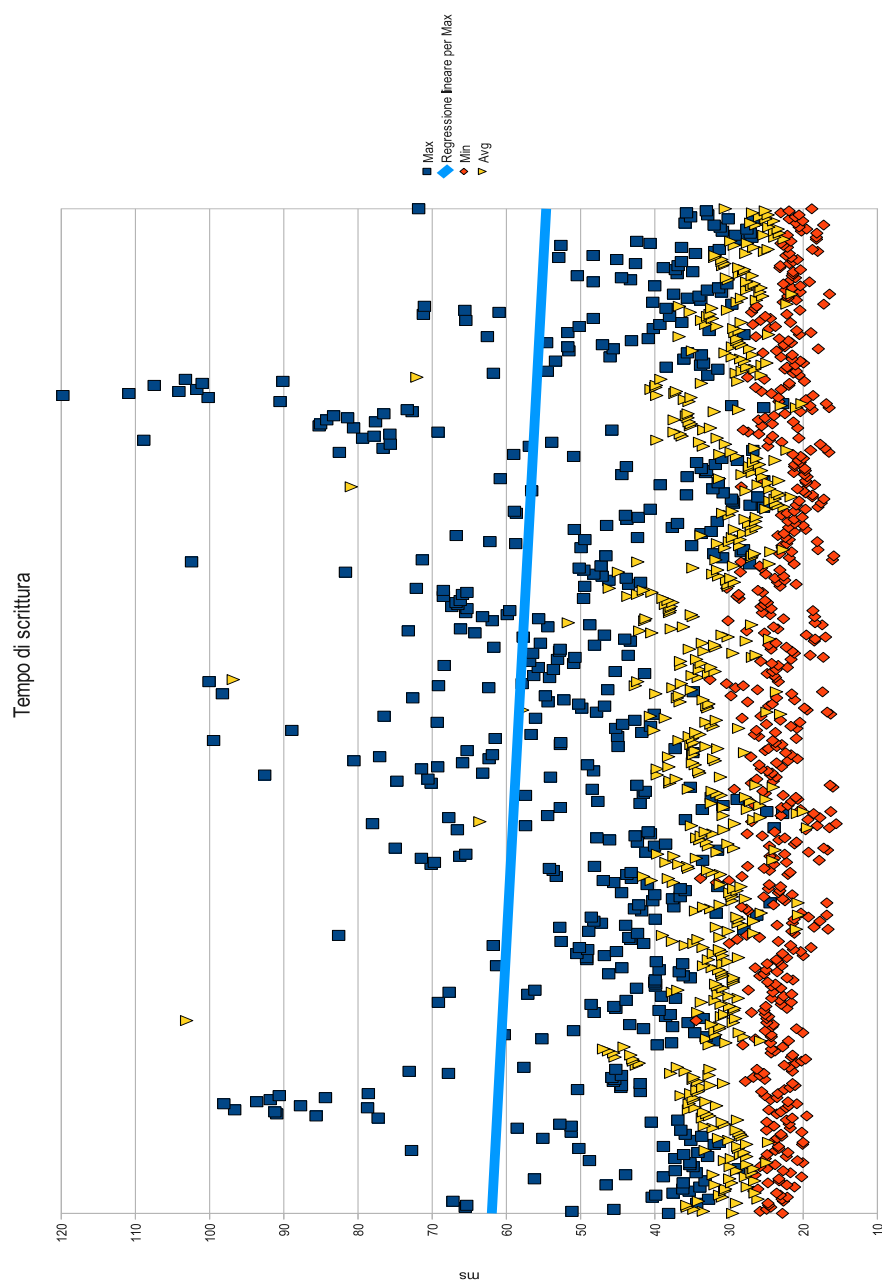
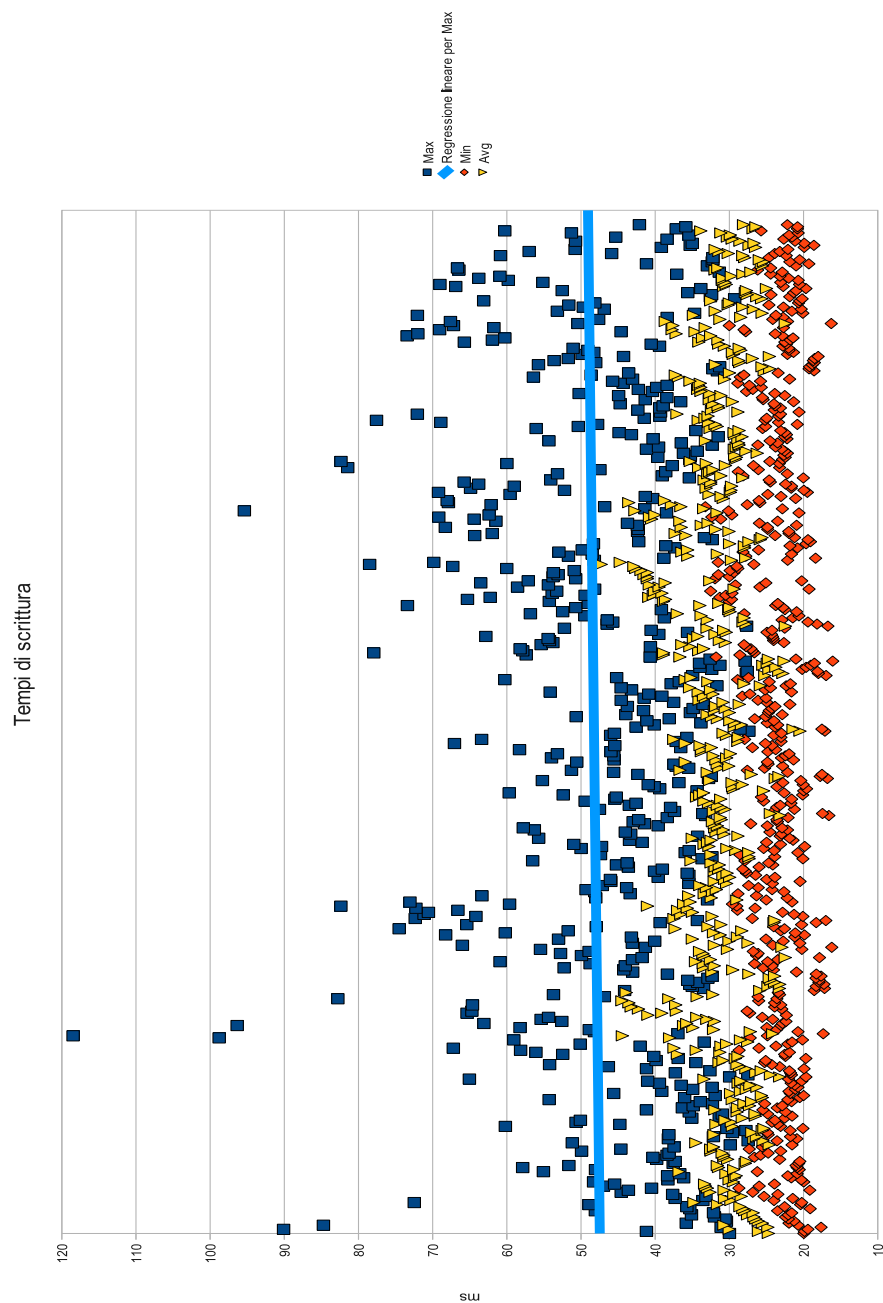
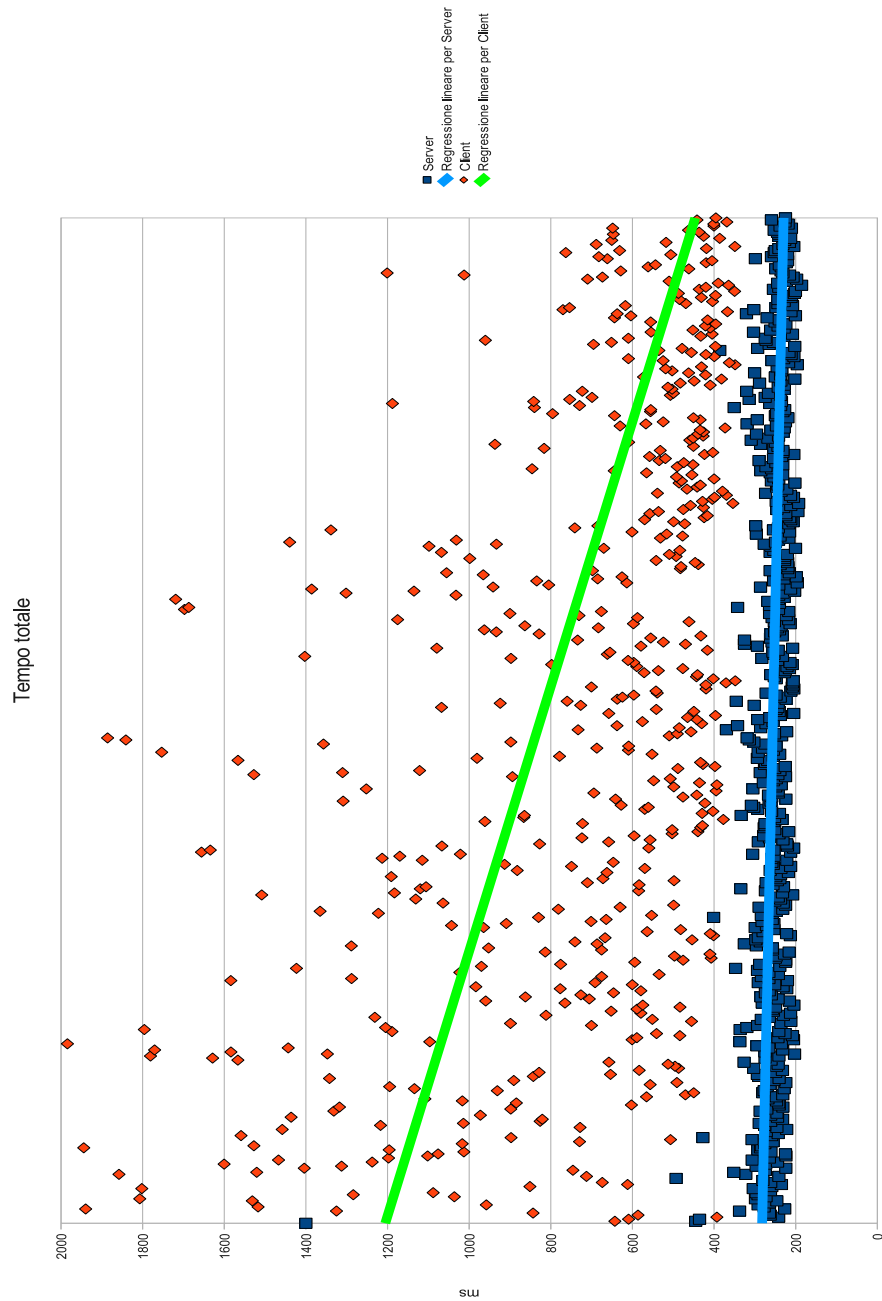


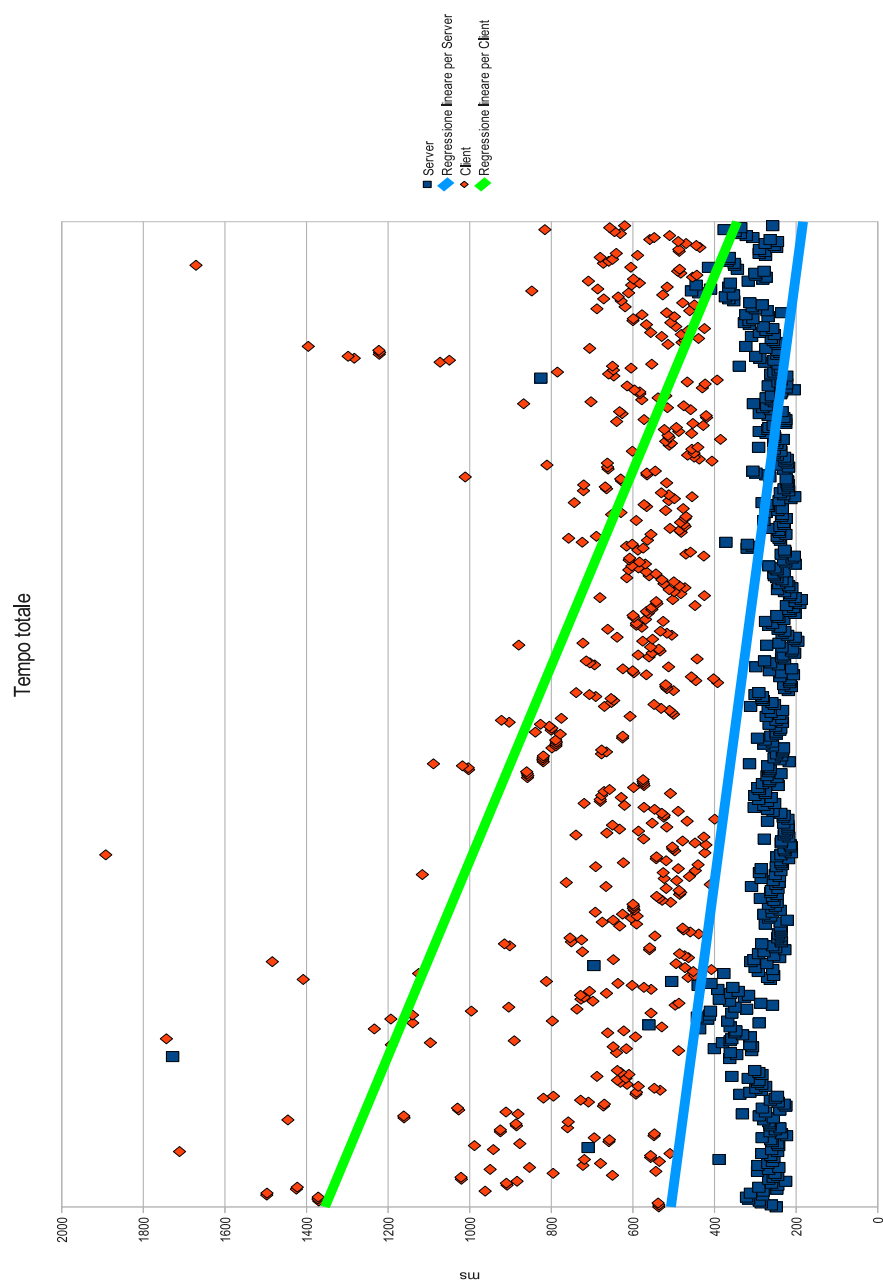
Figura B.11: Tempi di inserimento ($P=1$)

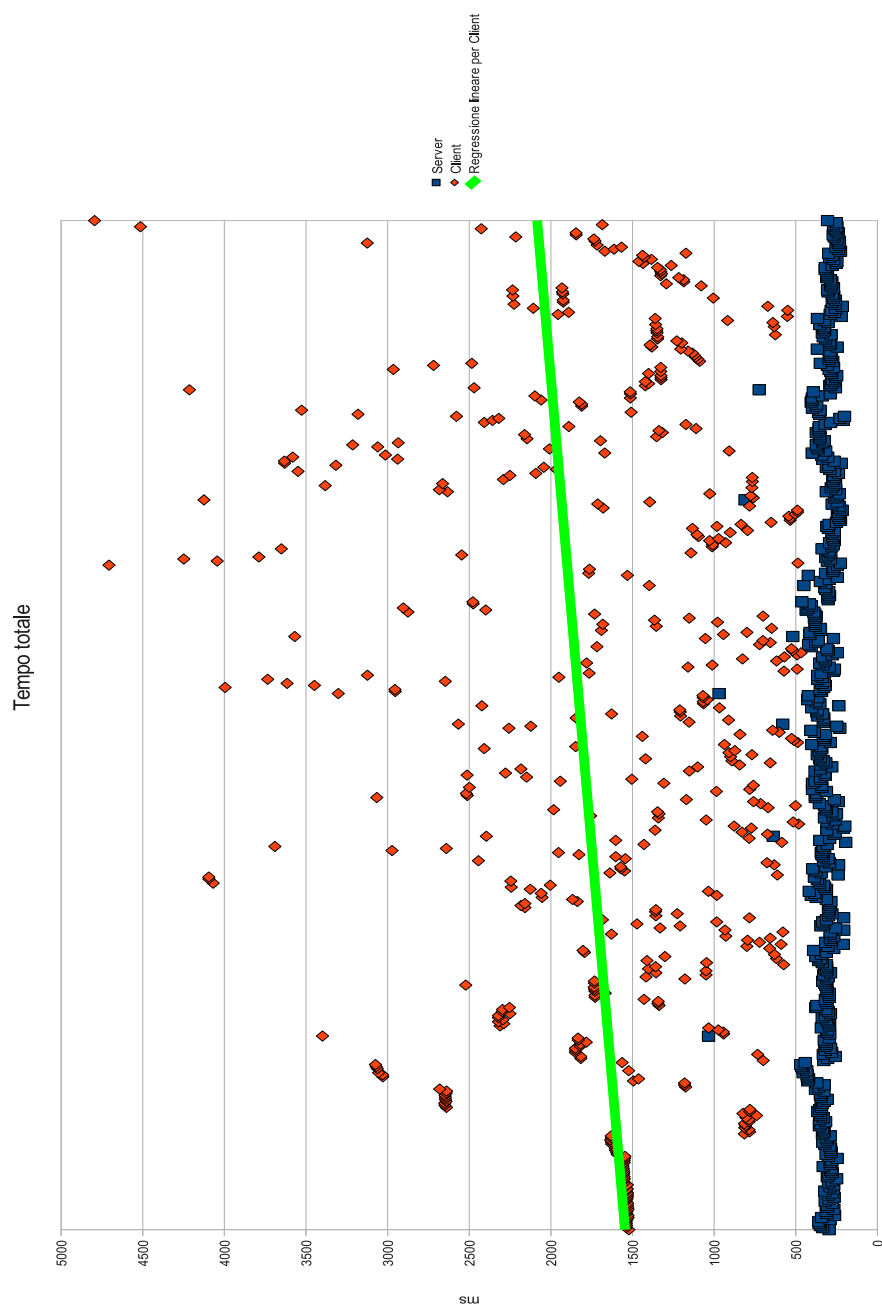
Figura B.12: Tempi di inserimento ($P=8$)

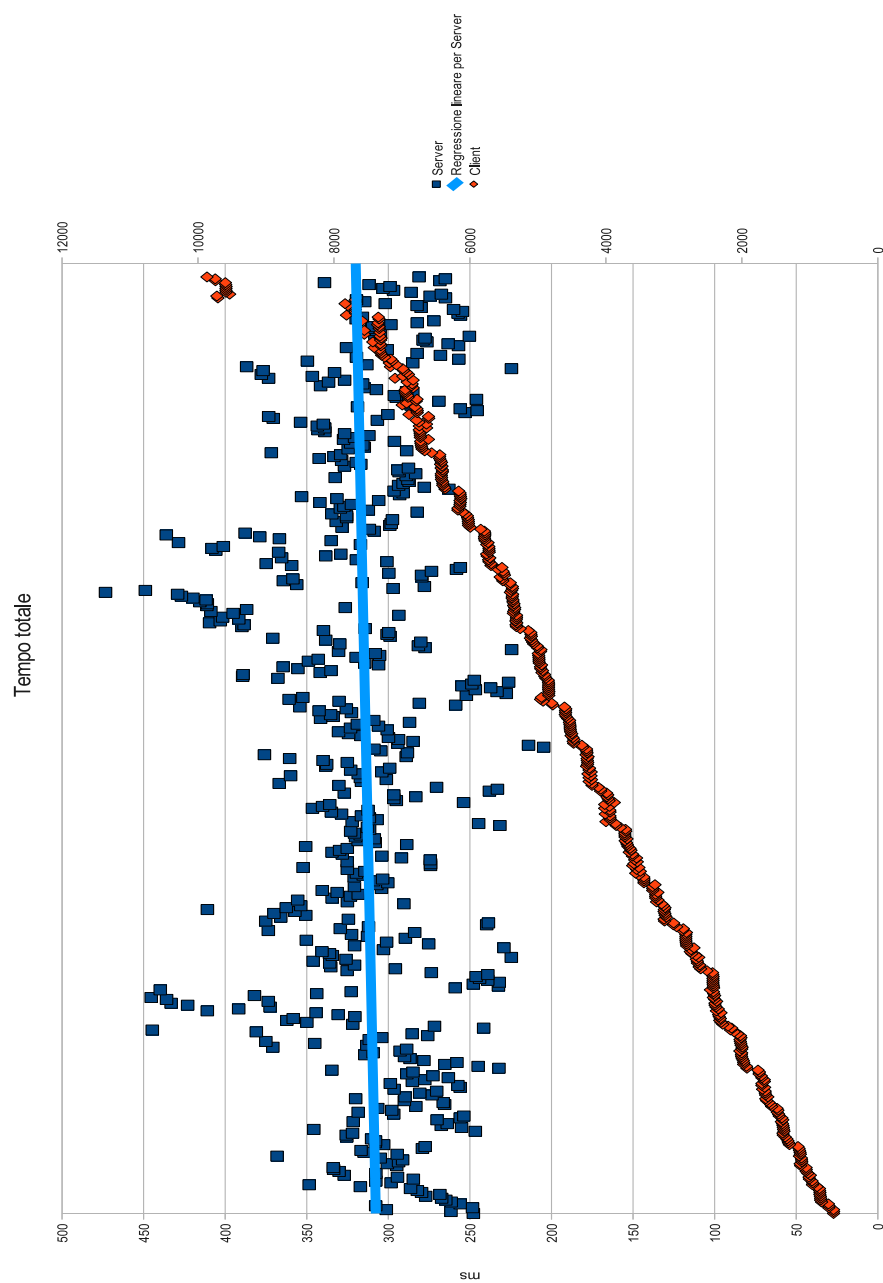
Figura B.13: Tempi di inserimento ($P=64$)

Figura B.14: Tempi di inserimento ($P=512$)

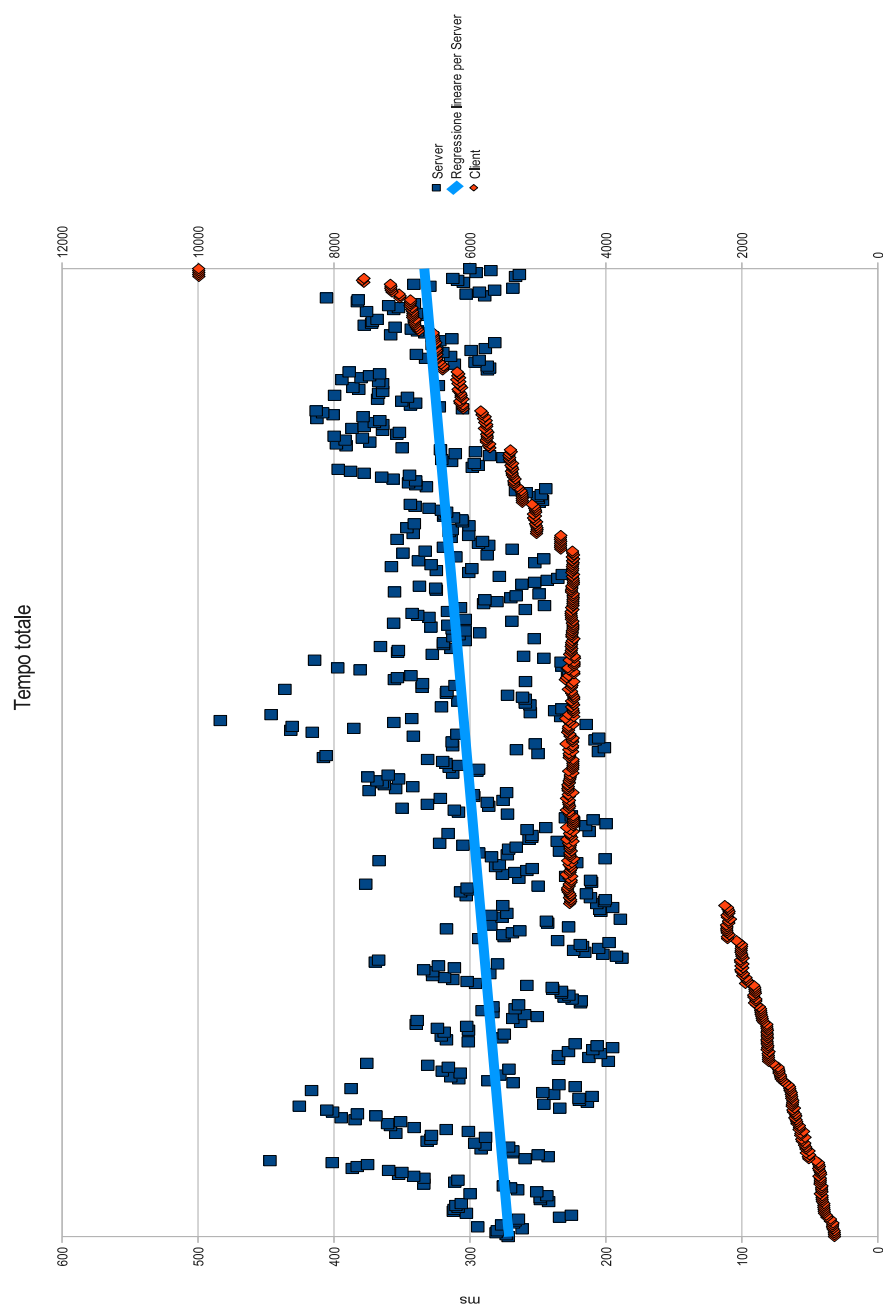
Figura B.15: Tempi totali di risposta ($P=1$)

Figura B.16: Tempi totali di risposta ($P=8$)

Figura B.17: Tempi totali di risposta ($P=64$)

Figura B.18: Tempi totali di risposta ($P=512$)

B.3 Problemi dovuti al proxy

Figura B.19: Tempi totali di risposta ($P=512$), per istanza

- [1] Mell Peter & Grance Tim (2009)
NIST Definition of Cloud Computing v15
<http://csrc.nist.gov/groups/SNS/cloud-computing/cloud-def-v15.doc>
- [2] NIST General Information
http://www.nist.gov/public_affairs/general_information.cfm
- [3] <http://www.salesforce.com/it/customers>
- [4] What is CRM?
<http://www.salesforce.com/uk/crm/what-is-crm.jsp>
- [5] What is AWS?
<http://aws.amazon.com/what-is-aws>
- [6] Products & Services
<http://aws.amazon.com/products>
- [7] <http://svn.apache.org/viewvc/jakarta/site/xdocs/images/logos>
- [8] Apache Tomcat
<http://tomcat.apache.org/index.html>
- [9] What is the Windows Azure platform?
<http://www.microsoft.com/windowsazure/>
- [10] EclipseIDE su Ubuntu Documentation
<https://help.ubuntu.com/community/EclipseIDE>
- [11] App Engine Product Roadmap
<http://code.google.com/intl/it-IT/appengine/docs/roadmap.html>
- [12] Developer's Guide su Google App Engine
<http://code.google.com/intl/it-IT/appengine/docs/>
- [13] Hypertext Transfer Protocol – HTTP/1.1
<http://tools.ietf.org/html/rfc2616>
- [14] Extensible Messaging and Presence Protocol (XMPP): Core
<http://tools.ietf.org/html/rfc3920>
- [15] Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence: Core
<http://tools.ietf.org/html/rfc3921>

- [16] Choosing a Datastore (Java)
<http://code.google.com/intl/it-IT/appengine/docs/java/datastore/hr/>
- [17] Chandra Tushar, Griesemer Robert & Redstone Joshua (2007)
Paxos Made Live – An Engineering Perspective
http://labs.google.com/papers/paxos_made_live.html
- [18] Chang Fay, Dean Jeffrey, Ghemawat Sanjay, Hsieh Wilson C., Wallach Deborah A., Burrows Mike, Chandra Tushar, Fikes Andrew & Gruber Robert E. (2006) @ OSDI'06: Seventh Symposium on Operating System Design and Implementation, WA, 2006
Bigtable: A Distributed Storage System for Structured Data
<http://labs.google.com/papers/bigtable.html>
- [19] Ghemawat Sanjay , Gobioff Howard & Leung Shun-Tak (2003) @ 19th ACM Symposium on Operating Systems Principles, NY, 2003
The Google File System
<http://labs.google.com/papers/gfs.html>
- [20] Simple Logging Facade for Java (SLF4J)
<http://www.slf4j.org/index.html>
- [21] JDO Enhancement
<http://db.apache.org/jdo/enhancement.html>
- [22] JPQL Language Reference
http://download.oracle.com/docs/cd/E16764_01/apirefs.1111/e13046/ejb3_langref.html
- [23] Interface HttpSession su Java EE 6 API Specification
<http://download.oracle.com/javaee/6/api/javax/servlet/http/HttpSession.html>
- [24] Enabling Sessions su GAE Developer's Guide
http://code.google.com/intl/it-IT/appengine/docs/java/config/appconfig.html#Enabling_Sessions
- [25] HTTP-sessions and the Google App Engine/J - some implementation details <http://blog.stringbuffer.com/2009/04/http-sessions-and-google-app-enginej.html>
- [26] Justin Haugh & Guido van Rossum (2011)
Scaling App Engine Applications @ Google I/O 2011
<http://www.google.com/events/io/2011/sessions/scaling-app-engine-applications.html>
- [27] Introducing JSON
<http://json.org/>

- [28] google-gson
<http://code.google.com/p/google-gson>