Apache Ignite: analisi di una piattaforma distribuita di elaborazione in-memory

Marco Meluzzi
marco.meluzzi@studio.unibo.it

Lorenzo Sutera
lorenzo.sutera@studio.unibo.it

17 aprile 2023

Abstract

Le aziende che vogliono perseguire iniziative di *digital transformation* spesso scoprono che la loro infrastruttura IT esistente non è in grado di supportare la velocità e la scalabilità richieste. Infatti, la necessità di fornire un'esperienza utente finale più personalizzata e in tempo reale costringe le aziende a trasformare i propri processi impostati in modalità batch, che possono richiedere ore oppure giorni, in processi automatizzati che operino in tempo reale.

L'adozione di tecnologie web, mobile, social o IoT per supportare questa trasformazione porta spesso a un incremento di interrogazioni e di transazioni fino a 10-1000 volte. La sfida è pertanto quella di riuscire a sfruttare a proprio vantaggio questo incremento nel volume dei dati, in modo da supportare in maniera più puntuale i processi decisionali senza però incorrere in tempi di elaborazione proibitivi.

L'*in-memory computing* è stata la risposta di molte aziende. L'elaborazione in memoria offre infatti velocità e scalabilità per applicazioni nuove ed esistenti. La velocità è resa possibile dal fatto che la memoria RAM viene utilizzata sia per l'archiviazione dei dati che per la loro elaborazione, evitando il più possibile di usare il disco il cui accesso è molto più lento. Dall'altra parte, la scalabilità viene raggiunta andando a distribuire sia i dati che le elaborazioni su un cluster di server distribuiti.

Tra le varie piattaforme che offrono questo tipo di caratteristiche e di prestazioni, Apache Ignite rappresenta un'ottima soluzione per i problemi sopracitati, trattandosi di una piattaforma open-source distribuita che sfrutta un cluster di più nodi a supporto dell'elaborazione, del caching e della persistenza dei dati, in modalità in-memory. Le sue funzionalità la rendono particolarmente adatta nell'ambito dell'*High-Performance Computing*, ed è usata da importanti aziende quali Microsoft, Netflix, Apple, Yahoo! e PayPal.

Indice

1 Introduzione

La natura di **Apache Ignite**[1] è duplice: da una parte costituisce una piattaforma distribuita di elaborazione, dall'altra funge anche da *database management system* (DBMS) distribuito. Entrambe le componenti sfruttano una soluzione *in-memory*, ossia operano direttamente in memoria centrale (RAM) per trarre vantaggio dalla sua intrinseca velocità, se paragonata ai tradizionali dispositivi di memoria di massa (*e.g.* HDD). Tale caratteristica rende Ignite un framework ideale per diversi contesti:

- applicazioni *cpu-intensive*, quali simulazioni scientifiche, previsioni meteo o modellazioni 3D, che necessitano di una solida infrastruttura di calcolo parallelo (e distribuito) che impieghi un cluster di computer (quindi tutto ciò che ricade nell'area della cosiddetta *High-Performance Computing*);
- applicazioni *data-intensive* che hanno bisogno di gestire enormi moli di dati, eventualmente anche in tempo reale, che devono quindi essere memorizzati, processati e analizzati in poco tempo (quindi tutto ciò che ricade nell'area dei *Big Data*);
- miglioramento di applicazioni esistenti, anche rivolte al mercato consumer, per le quali si vogliono ridurre i costi di produzione e di deployment e al tempo stesso garantire agli utenti un'esperienza di accesso e di utilizzo più veloce e performante (*e.g.* applicazioni e/o servizi resi disponibili sul Web a utenti finali o aziende).

La Figura 1 schematizza quella che in genere è l'architettura di una classica applicazione che comprende una parte *front-end* (*i.e.* dei client che si connettono ad un certo servizio), una parte *back-end* (*i.e.* i server su cui è in esecuzione il servizio) e un *data store* su cui vengono memorizzati in modo persistente i dati.

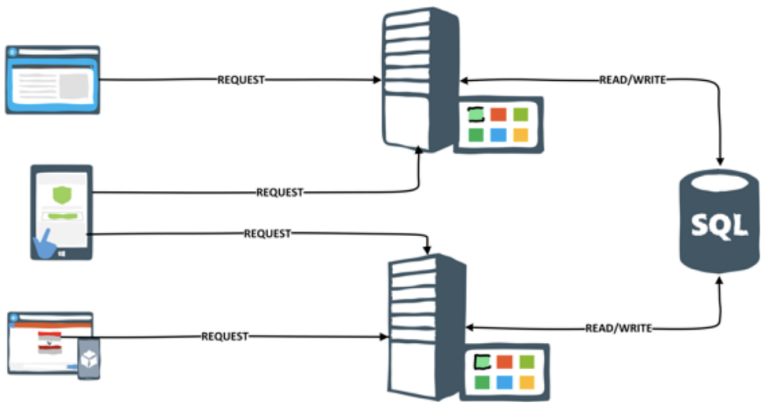

Figura 1: Tradizionale architettura di un'applicazione client-server

Tipicamente, qualsiasi interazione di lettura/scrittura verso quest'ultimo componente avviene in maniera sincrona, garantendo consistenza e durabilità dei dati. Tuttavia, nel caso sopraggiunga un alto volume di richieste transazionali, può verificarsi facilmente

un collo di bottiglia, con conseguente degrado delle prestazioni. Per far fronte a questo problema, è possibile aggiungere all'architettura mostrata sopra un ulteriore layer tra i server e il data store che lavori esclusivamente in memoria RAM. Tale componente aggiuntivo può quindi essere sfruttato ad esempio per memorizzare i dati acceduti più di frequente dai client, senza dovere ogni volta contattare il data store che verrà interpellato solo in caso di necessità (*e.g.* per inviare o ricevere aggiornamenti), possibilmente in maniera asincrona. Considerato che i dati si spostano vicino agli endpoint dell'applicazione, questo approccio può ridurre sensibilmente il tempo di risposta e diminuire così il tempo necessario alle transazioni. Tale architettura, esemplificata in Figura 2, può essere facilmente implementata tramite Ignite.

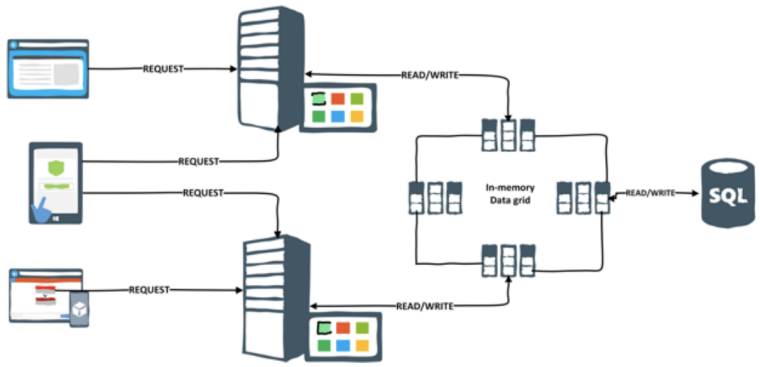

Figura 2: Architettura in-memory di un'applicazione client-server

In Figura 3 sono mostrate alcune delle principali feature messe a disposizione da Ignite. La presenza di determinate funzionalità, assenti in soluzioni competitor alternative, assieme alla facilità di utilizzo del framework, ha reso Ignite popolare in svariati contesti aziendali di rilievo, oltre che nell'implementazione di diversi casi d'uso, anche in scenari di ricerca.

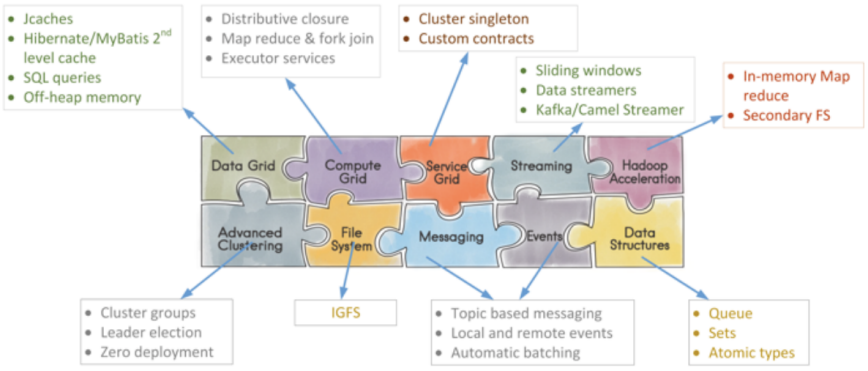

Figura 3: Alcune feature offerte da Apache Ignite

2 Architettura

2.1 Architettura fisica

Il funzionamento di Apache Ignite[2] si basa sull'utilizzo di nodi, organizzati in uno o più cluster. Tali componenti adottano un'architettura *shared-nothing*, il che significa che non condividono alcuna risorsa ma che ciascuno è equipaggiato con il proprio processore, memoria RAM e disco. In questo modo, viene eliminata qualsiasi situazione di contesa tra i nodi, i quali invece collaborano a risolvere un problema comune suddividendolo in diversi *task* indipendenti. Il principale vantaggio di una tale architettura risiede nel fatto di eliminare i cosiddetti *single point of failure*, ovvero quelle componenti che in caso di fallimento causano il blocco dell'intero sistema. In questo scenario, invece, tutto continua a funzionare senza interruzioni se un singolo nodo dovesse fallire, essere rimosso o aggiunto.

A differenza di altre tecnologie e framework, che si basano su architetture monolitiche o di tipo *master-slave* (*e.g.* Apache Hadoop), Ignite non necessita di un nodo master che coordini e guidi gli altri nodi, dal momento che questi vengono trattati come entità di pari livello con le stesse caratteristiche e funzioni.

Nonostante ciò, è possibile assegnare a ciascun nodo uno tra i seguenti due ruoli:

- **nodo server**: memorizza i dati, partecipa al caching, alla computazione, all'elaborazione streaming e può essere coinvolto nell'esecuzione dei task di *MapReduce* in-memory (vedi Capitolo 3.1).
- **nodo client**: permette di collegarsi da remoto ai server per effettuare operazioni di inserimento/lettura di elementi nella/dalla cache e per eseguire le query. Può inoltre memorizzare porzioni di dati in *near cache*, una piccola cache locale che memorizza i dati acceduti più frequentemente e più di recente (vedi Capitolo 2.2.1).

2.1.1 Nodi server

Se non esplicitato diversamente, tutti i nodi Ignite vengono avviati di default in modalità server. Come già accennato sopra, i nodi server che compongono il cluster possono comunicare direttamente tra di loro, senza l'ausilio di un coordinatore. Infatti, essi fungono congiuntamente da *data note* e da *compute node*, dal momento che sono responsabili sia della memorizzazione dei dati che della loro effettiva computazione.

In base a come viene gestito il deployment dei nodi server, possono presentarsi i seguenti scenari:

- **server incorporato con l'applicazione**: con questo approccio, il nodo server viene eseguito sulla stessa JVM in cui si trova l'applicazione (*e.g.* applicazione web o applicazione Java standalone) ed è in grado di unirsi agli altri nodi della griglia, secondo lo schema raffigurato in Figura 4 della pagina seguente. In questo modo, quando l'applicazione porta a termine il proprio task, o quando fallisce, anche il server viene spento.

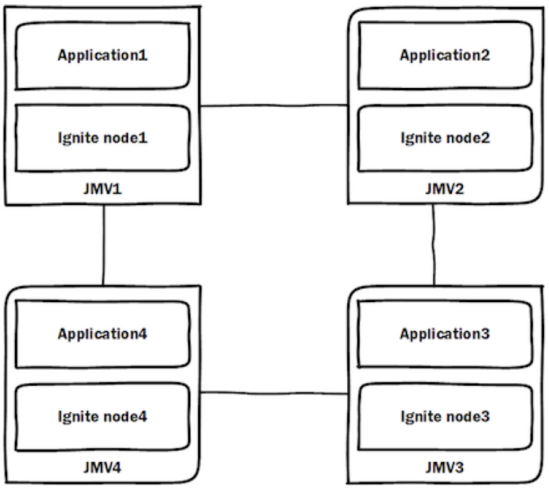

Figura 4: Server incorporato con l'applicazione

- **server separato dall'applicazione**: in questo caso, i nodi server vengono eseguiti su una JVM separata rispetto alle applicazioni, che si connettono in remoto ai server. Tale approccio, schematizzato in Figura 5, è quello più comunemente usato e che fornisce la maggiore flessibilità possibile, dal momento che ciascun server può essere spento e riavviato senza impattare sull'esecuzione dell'applicazione o sul resto del cluster.

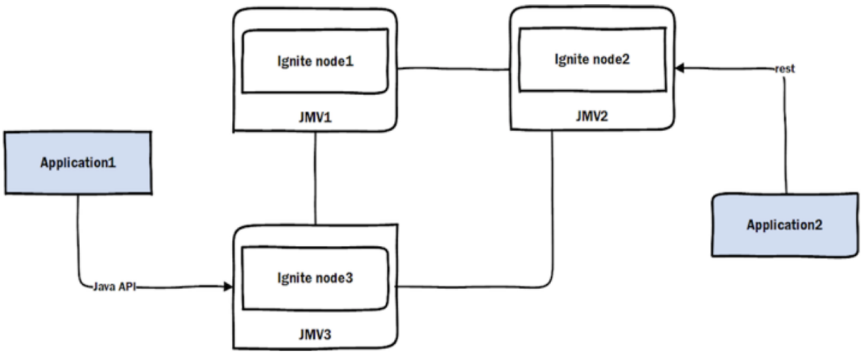

Figura 5: Server separato dall'applicazione

2.1.2 Nodi client

In uno scenario classico, lo scopo dei nodi client è unicamente quello di connettersi in remoto ai nodi server per ottenere i dati qui memorizzati o aggiornarvi il contenuto. Qualsiasi altro tipo di computazione viene demandata ai server.

Tuttavia, in certi casi si può decidere che anche i client possano partecipare ai task computazionali, manipolando il contenuto delle cache dei server e memorizzando i propri dati in locale. Ad esempio, nel caso di un alto volume di transazioni ACID, si potrebbe decidere di eseguire i task sui client, creando un corrispondente gruppo di nodi. Così facendo, si separano i compute node (questa volta rappresentati dai nodi client) dai data node (rappresentati dai nodi server) e si viene a formare un'architettura come quella mostrata in Figura 6.

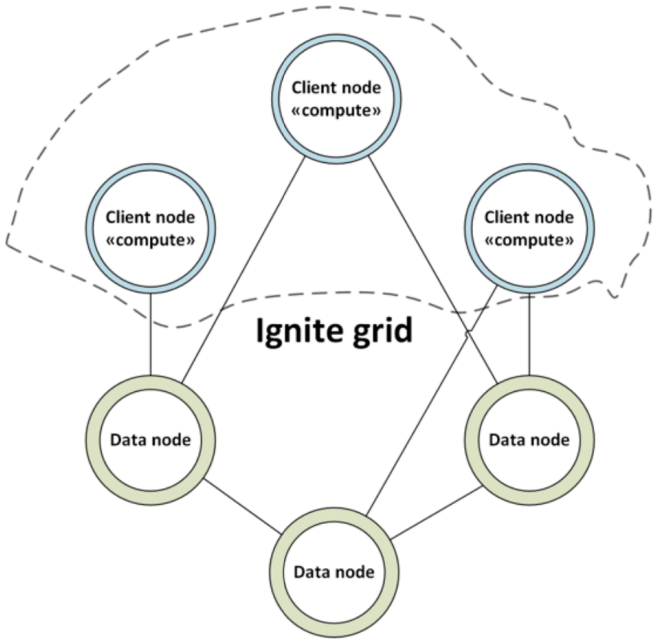

Figura 6: Architettura con data note e compute node separati

Tale approccio presenta però degli svantaggi: contraddicendo la filosofia alla base di Ignite, secondo cui ciascun nodo server è responsabile sia della memorizzazione dei dati che dell'esecuzione dei task computazionali, i dati vengono allocati su nodi separati, perciò per la loro computazione i client devono recuperarli dai nodi server. Ciò può produrre molte connessioni di rete e portare a una latenza non tollerabile.

È possibile altresì distinguere tra due diverse tipologie di nodi client:

- **thick client**: assume il comportamento descritto sinora, con la possibilità di connettersi da remoto ai nodi server, oltre a poter gestire opzionalmente anche la computazione, lo streaming, i servizi e il near caching. Maggiori dettagli su queste funzionalità verranno forniti nel prosieguo della trattazione.

- **thin client**: è una sorta di client *lightweight* che si connette al cluster tramite una connessione socket standard. Non entra a far parte della topologia del cluster, quindi non memorizza nessun tipo di dato e non prende parte ad alcun tipo di

computazione. Tutte le operazioni che definisce vengono perciò passate (ed eseguite) al nodo server con cui ha stabilito la connessione. In caso di errore del nodo o della connessione, il client è in grado di passare automaticamente ad un altro nodo disponibile, grazie alla presenza di un meccanismo di *connection failover*.

Una feature esclusiva di questo tipo di client è la possibilità di attivare la *partition awareness*. Come si avrà modo di spiegare successivamente (vedi Capitolo 2.4.2), i dati sono suddivisi tra i diversi nodi server. Con tale funzionalità, il thin client può quindi instradare le query e le operazioni direttamente verso il nodo che possiede i dati necessari, eliminando qualsiasi collo di bottiglia e favorendo la scalabilità dell'applicazione. In caso contrario, il thin client esegue le operazioni attraverso un singolo nodo server che funge da proxy per tutte le richieste in entrata e che può quindi facilmente diventare un collo di bottiglia, dovendo gestire anche il re-instradamento di ciascuna operazione verso il nodo specifico che memorizza i dati richiesti. La Figura 7 mostra la differenza di comportamento con e senza partition awareness.

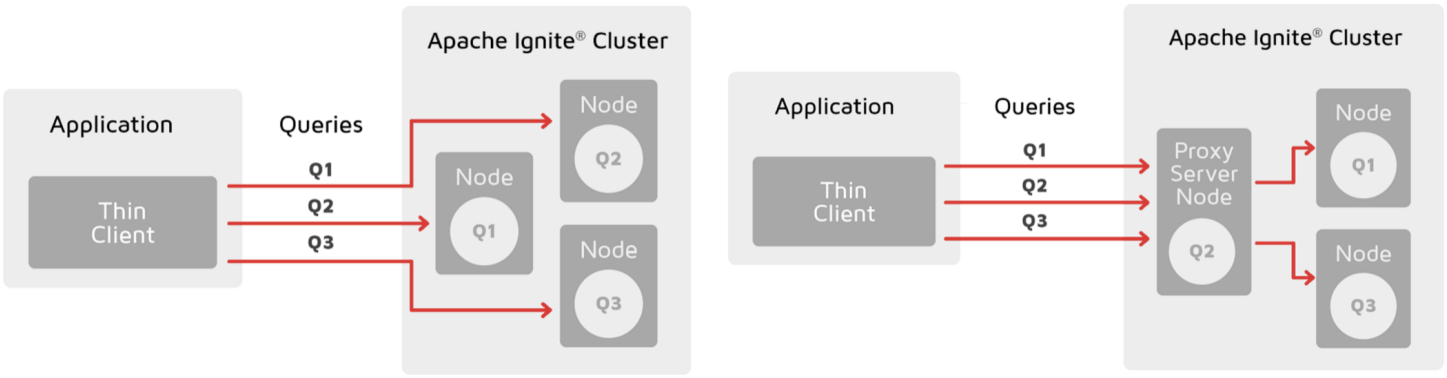

Figura 7: Thin client con e senza partition awareness

2.1.3 Clustering topology

All'avvio, a ciascun nodo viene assegnato il ruolo di client o di server. La peculiarità di quest'ultimo tipo di nodo risiede nel fatto che una volta creato esso si unisce direttamente al cluster, cioè è in grado di trovare e connettersi automaticamente a tutti gli altri data node, che si dispongono in una topologia ad anello, come mostrato in Figura 8 della pagina seguente. Ciò consente ad Ignite di poter scalare al bisogno, semplicemente aggiungendo ulteriori nodi senza dover riavviare l'intero cluster, il quale può crescere orizzontalmente in maniera potenzialmente infinita. Questa importante proprietà di *elasticità* è cruciale per un sistema distribuito.

L'insieme di nodi server che si viene a creare avrà il compito di effettuare le computazioni richieste dai client, in maniera distribuita, bilanciata e *fault-tolerant*. Possono essere inviati singoli task computazionali così come può essere implementato il pattern

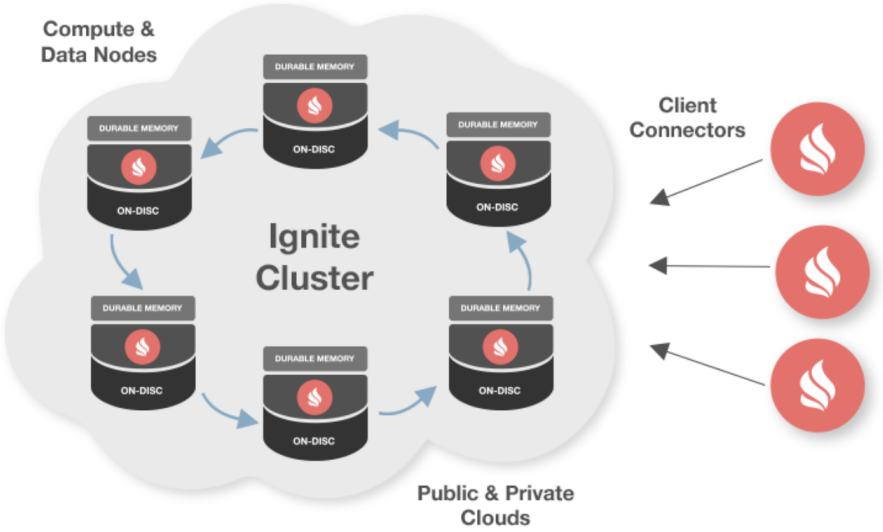

Figura 8: Topologia di un cluster Ignite

MapReduce con la suddivisione automatica dei task. Allo stesso modo, anche i dati sono distribuiti e bilanciati tra i nodi del cluster.

Cluster groups I nodi server possono anche essere raggruppati in *cluster group*. Si tratta di una suddivisione logica che permette di limitare lo scope di specifiche operazioni ad un sottoinsieme di nodi, piuttosto che utilizzare l'intero cluster.

Sostanzialmente, un cluster group può essere creato a partire da:

- cluster group predefiniti da Ignite (*e.g.* nodi remoti);
- nodi con attributi definiti dall'utente (ai quali si vuole associare una specifica semantica);
- nodi che soddisfano un qualche predicato (in questo caso il gruppo definito sarà dinamico e potrà cambiare nel tempo in base al mutare di certe condizioni).

Per i dettagli implementativi sulla creazione dei gruppi, fare riferimento al Capitolo 5.1.

Meccanismi di discovery Affinché i nodi possano scoprirsi automaticamente tra di loro e formare il cluster, è però necessario configurare un meccanismo di *discovery* appropriato. Di default, Ignite utilizza il **TCP/IP Discovery**, progettato e ottimizzato per gestire un centinaio di nodi. Tuttavia, nel caso in cui si debba scalare da qualche centinaio a qualche migliaio di nodi, la struttura stessa ad anello che caratterizza la topologia del cluster inizia ad essere limitante, dal momento che un messaggio di sistema

impiegherà del tempo a passare attraverso tutti i nodi e di conseguenza la risposta a certi eventi, quali l'aggiunta di un nuovo nodo o il rilevamento di uno che ha fallito, sarà ritardata. Tutto ciò influisce sulla reattività e sulle prestazioni complessive del cluster.

In questi casi, per cercare di preservare una scalabilità lineare e delle buone performance, è opportuno utilizzare lo **ZooKeeper Discovery**. Tale meccanismo fa uso di ZooKeeper[1] come singolo punto di sincronizzazione, organizzando i nodi in una topologia a stella dove il cluster ZooKeeper si trova al centro e i nodi Ignite si scambiano eventi di discovery attraverso di esso, come schematizzato in Figura 9.

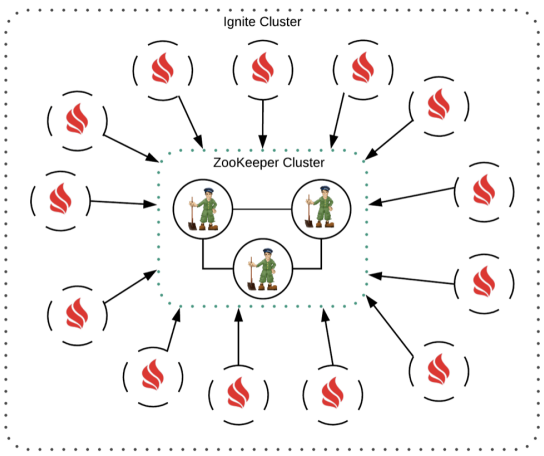

Figura 9: Cluster con topologia a stella che si viene a formare con l'uso di ZooKeeper Discovery

Lo ZooKeeper Discovery offre supporto anche in caso di *network partitioning* o nel caso di fallimento di comunicazione tra nodi. Infatti, si assume che il cluster ZooKeeper sia sempre visibile a tutti i nodi del cluster e nell'eventualità in cui un nodo si disconnetta da ZooKeeper esso verrà spento e quindi trattato come tale dagli altri nodi.

Appena un nodo scopre che non riesce a connettersi a qualche altro nodo del cluster, fa partire un processo di risoluzione della failure di comunicazione pubblicando delle richieste speciali al cluster ZooKeeper. Tutti i nodi tentano quindi di connettersi gli uni agli altri, inviando i risultati di questi tentativi al nodo che per l'occasione funge da coordinatore del processo. Sulla base di queste informazioni, il nodo coordinatore crea un grafo delle connessioni che rappresenta la situazione di rete del cluster. In base al tipo di segmentazione che si verifica, possono delinearsi i seguenti scenari:

- se il cluster Ignite si divide in più componenti indipendenti, ciascuno di essi, in quanto cluster a tutti gli effetti, continua a processare le richieste degli utenti senza sapere di non avere una visione completa della situazione, e ciò può portare all'inconsistenza dei dati. Per evitare questa situazione, solo il cluster con il maggior

[1] Apache ZooKeeper[3] è un servizio di coordinamento distribuito per applicazioni distribuite, che offre supporto nella gestione della comunicazione tra i diversi host.

numero di nodi viene lasciato attivo, mentre tutti gli altri nodi vengono spenti. Se si verifica una situazione di parità per quanto riguarda la dimensione dei diversi cluster, viene mantenuto attivo quello che ha il maggior numero di client connessi. La Figura 10 esemplifica tale scenario.

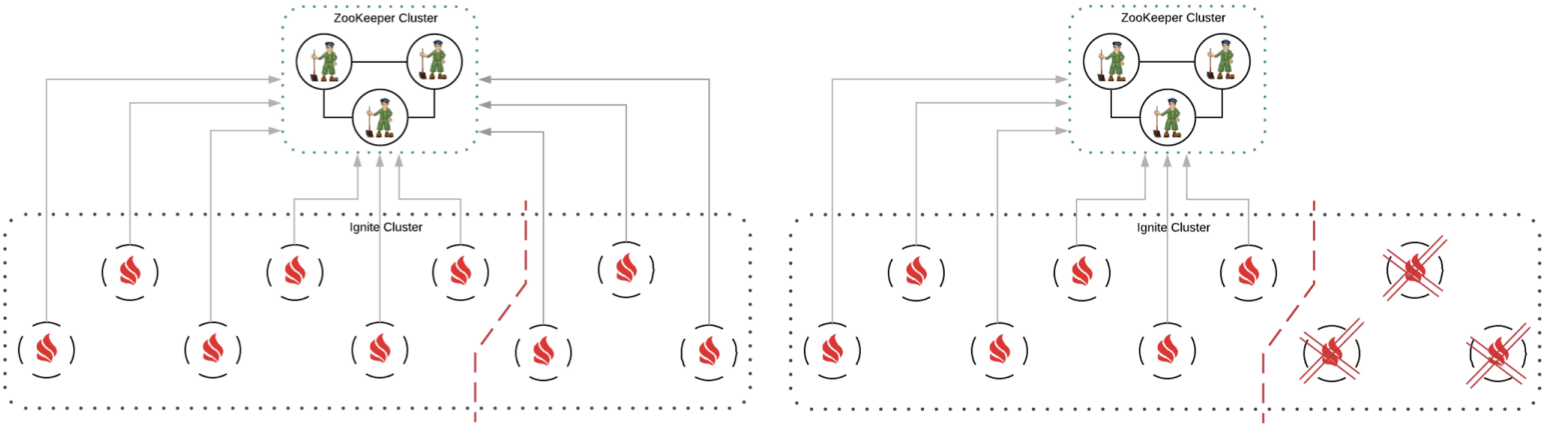

Figura 10: In caso di partizione di rete, i nodi del cluster più piccolo (a destra) vengono spenti

- se dei collegamenti sono mancanti, allora alcuni nodi non riescono a connettersi a certi altri nodi, il che significa che pur non essendo completamente disconnessi dal cluster non riescono a scambiare dati con tutti. La soluzione, che consiste nel trovare la più grande componente nella quale ogni nodo può connettersi a tutti gli altri, è un problema complesso che non può essere risolto in tempi ragionevoli. Il nodo coordinatore usa pertanto un algoritmo euristico che cerca di trovare la migliore soluzione approssimata. I nodi che non fanno parte della componente trovata vengono spenti. In Figura 11 è riportata questa situazione.

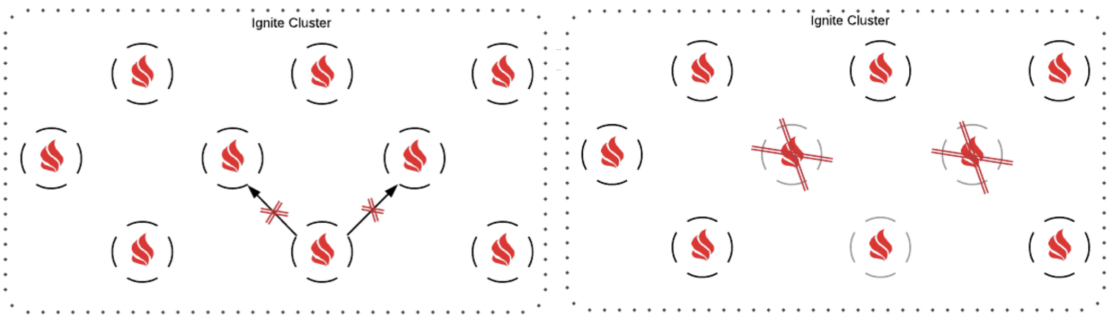

Figura 11: A sinistra, un nodo non riesce a connettersi agli altri due nodi, a destra tali nodi vengono spenti non facendo parte della più larga componente trovata dall'algoritmo euristico

- nei deployment su larga scala, può capitare che il cluster ZooKeeper sia suddiviso su più data center, eventualmente distribuiti geograficamente. In caso di partizione di rete, quindi, si vengono a creare segmenti multipli, tra i quali ZooKeeper controlla che ce ne sia uno che contenga più della metà di tutti i nodi ZooKeeper, condizione necessaria affinché possa operare. Se viene trovato, tale segmento prende il controllo del cluster Ignite, in caso contrario ZooKeeper spegne tutti i suoi nodi. Parallelamente alla segmentazione del cluster ZooKeeper, può verificarsi anche quella del cluster Ignite. In qualunque caso, quando i nodi ZooKeeper vengono spenti, i nodi Ignite ad esso associati cercano di connettersi ai nodi ZooKeeper rimasti, spegnendosi a loro volta se tale operazione non ha successo.

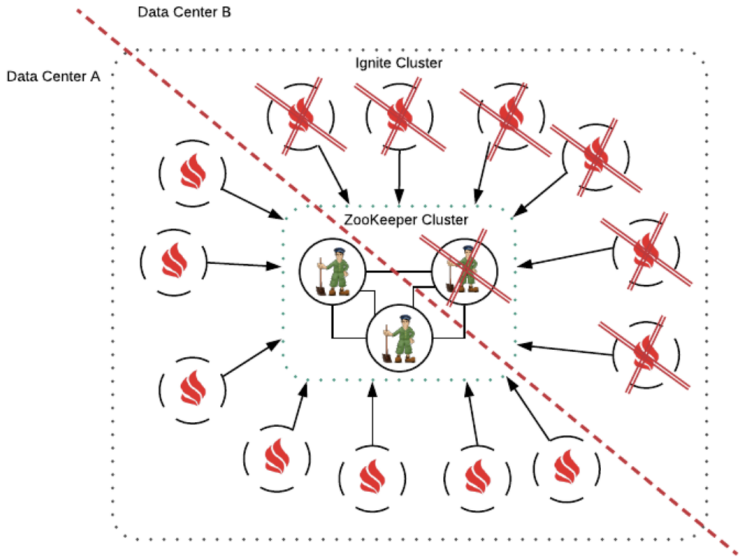

Figura 12: Segmentazione del cluster ZooKeeper e del corrispondente cluster Ignite

La Figura 12 mostra la situazione in cui una partizione di rete divide entrambi i cluster in due segmenti, eventualità riscontrabile nel caso in cui i due cluster siano distribuiti su due data center. Di conseguenza, il nodo ZooKeeper che si trova sul data center B termina (essendo il segmento più piccolo) e i nodi Ignite dello stesso data center terminano a loro volta in quanto non vi sono più nodi ZooKeeper disponibili a cui connettersi nel segmento in questione.

È bene ricordare che l'uso congiunto di Ignite e ZooKeeper richiede di configurare e gestire due sistemi distribuiti indipendenti, il che può essere arduo, pertanto è consigliabile passare allo ZooKeeper Discovery solo quando effettivamente necessario.

Per i dettagli su come configurare opportunamente un meccanismo di discovery e i relativi parametri di rete, fare riferimento al Capitolo 5.2.

Baseline topology Tenere traccia dell'insieme dei nodi che sono responsabili della memorizzazione dei dati è fondamentale per garantire la possibilità di gestire il corretto bilanciamento dei dati nel cluster (vedi Capitolo 2.4.5). Il concetto di *baseline topology* è proprio stato introdotto allo scopo di controllare questo processo ed evitare trasferimenti di dati non necessari quando ad esempio un nodo server lascia per un breve periodo di tempo il cluster, situazione che può verificarsi a causa di interruzioni occasionali di rete o per una manutenzione programmata del nodo. La baseline topology include unicamente i nodi server, dal momento che sono gli unici che possono memorizzare dati.

Quando la feature di regolazione automatica della baseline topology è abilitata, il cluster è in grado di monitorare lo stato dei suoi nodi server e di impostare automaticamente la baseline della topologia corrente quando questa rimane stabile per un certo periodo di tempo configurabile (di default 5 minuti). Se nel frattempo l'insieme di nodi del cluster cambia, il timeout viene aggiornato. Il timer è impostato di default a 0 per i cluster che operano esclusivamente in-memory (vedi Capitolo 2.2.2), che quindi lanciano il rebalancing ad ogni cambiamento della topologia. Nel caso la persistenza sia abilitata (vedi Capitolo 2.2.3), il cluster risulta essere inattivo la prima volta che viene avviato, stato in cui tutte le operazioni sono proibite. A seguito dell'attivazione manuale del cluster, l'insieme corrente di nodi viene designato come baseline topology: da questo momento in poi, se il cluster viene riavviato è in grado di attivarsi automaticamente non appena tutti i nodi registrati nella baseline si collegano. Tuttavia, se per qualche ragione certi nodi non riescono ad unirsi al cluster dopo un riavvio, è necessario attivarlo nuovamente a mano.

2.2 Architettura di memoria

Come si ha già avuto modo di anticipare, il principale punto di forza di Apache Ignite risiede nella possibilità di eseguire ogni sorta di computazione e di gestire la persistenza dei dati direttamente in memoria centrale. Ciononostante, all'occorrenza è anche possibile scegliere tra differenti modalità di utilizzo che sfruttano invece la durabilità del disco.

Tale archiviazione a più livelli, raffigurata in Figura 13 della pagina successiva, funziona in modo simile alla memoria virtuale dei sistemi operativi, come Linux. Tuttavia, una differenza significativa tra questi due tipi di architettura è che lo storage multi-livello di Ignite, nel caso sia abilitata la persistenza, tratta il disco come un superset dei dati, in grado di sopravvivere a crash e riavvii del sistema, mentre la memoria virtuale tradizionale utilizza il disco solo come estensione di swap, che viene cancellata una volta terminato il processo.

L'architettura multi-livello è basata sul concetto di *pagina*, che rappresenta l'unità fondamentale in cui viene suddivisa l'intera memoria. Queste pagine hanno una dimensione fissa e vengono memorizzate in specifiche regioni della memoria RAM chiamate *off-heap*, dal momento che si trovano al di fuori dello heap Java. Oltre a memorizzare le data entry, tipicamente si rende necessario mantenere nelle pagine anche degli indici che permettano di ottimizzare l'esecuzione di determinate query SQL.

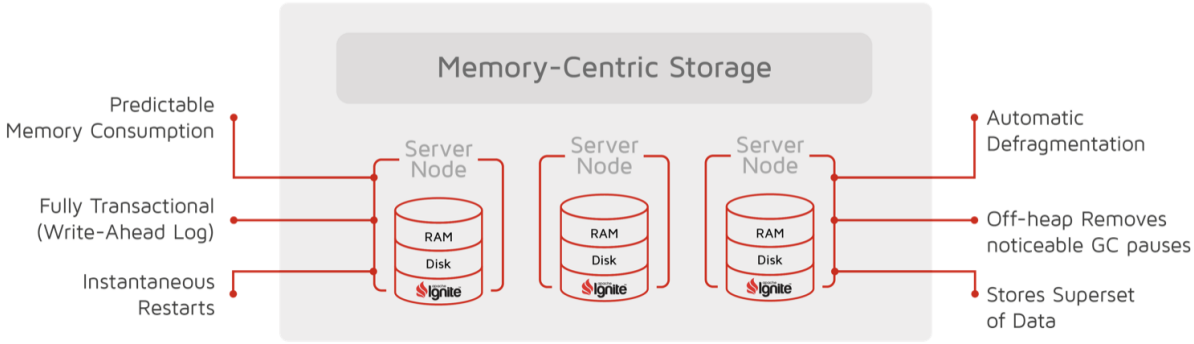

Figura 13: Architettura di storage multi-livello di un cluster Ignite

Ignite mantiene la stessa rappresentazione binaria dei dati sia in memoria che su disco, in modo da eliminare la necessità di costose serializzazioni quando ci si sposta da una all'altro.

La Figura 14 della pagina seguente mostra nel dettaglio come è strutturata l'architettura multi-livello: la memoria RAM è organizzata in *data region*, a ciascuna delle quali viene assegnata una dimensione iniziale e una dimensione massima verso cui può crescere allocando segmenti di memoria continua. Un segmento di memoria può essere visto come un array di byte continuo diviso in pagine di dimensione fissata. Esiste una data region attiva di default su ciascun nodo, che può richiedere fino al 20% della RAM disponibile, in cui vengono allocate tutte le cache create, ma ci sono situazioni in cui è necessario crearne di aggiuntive, ad esempio se si vogliono configurare delle cache con la persistenza attiva. Per maggiori dettagli su come configurare le data region, fare riferimento al Capitolo 5.3.

Le pagine sono dotate di un identificativo attraverso cui è possibile linkarle tra di loro e costruire delle dipendenze reciproche che danno origine a delle strutture complesse ad albero, implementate attraverso dei B+Tree. Se la persistenza è abilitata, in seguito a un restart Ignite potrà caricare il nodo radice che contiene i metadati necessari per iterare sugli altri livelli dell'albero e quindi risalire alle pagine mancanti in memoria e caricarle dal disco. Inoltre, le pagine dell'albero possono essere unite assieme se meno del 50% dello spazio della pagina viene usato dai dati di payload.

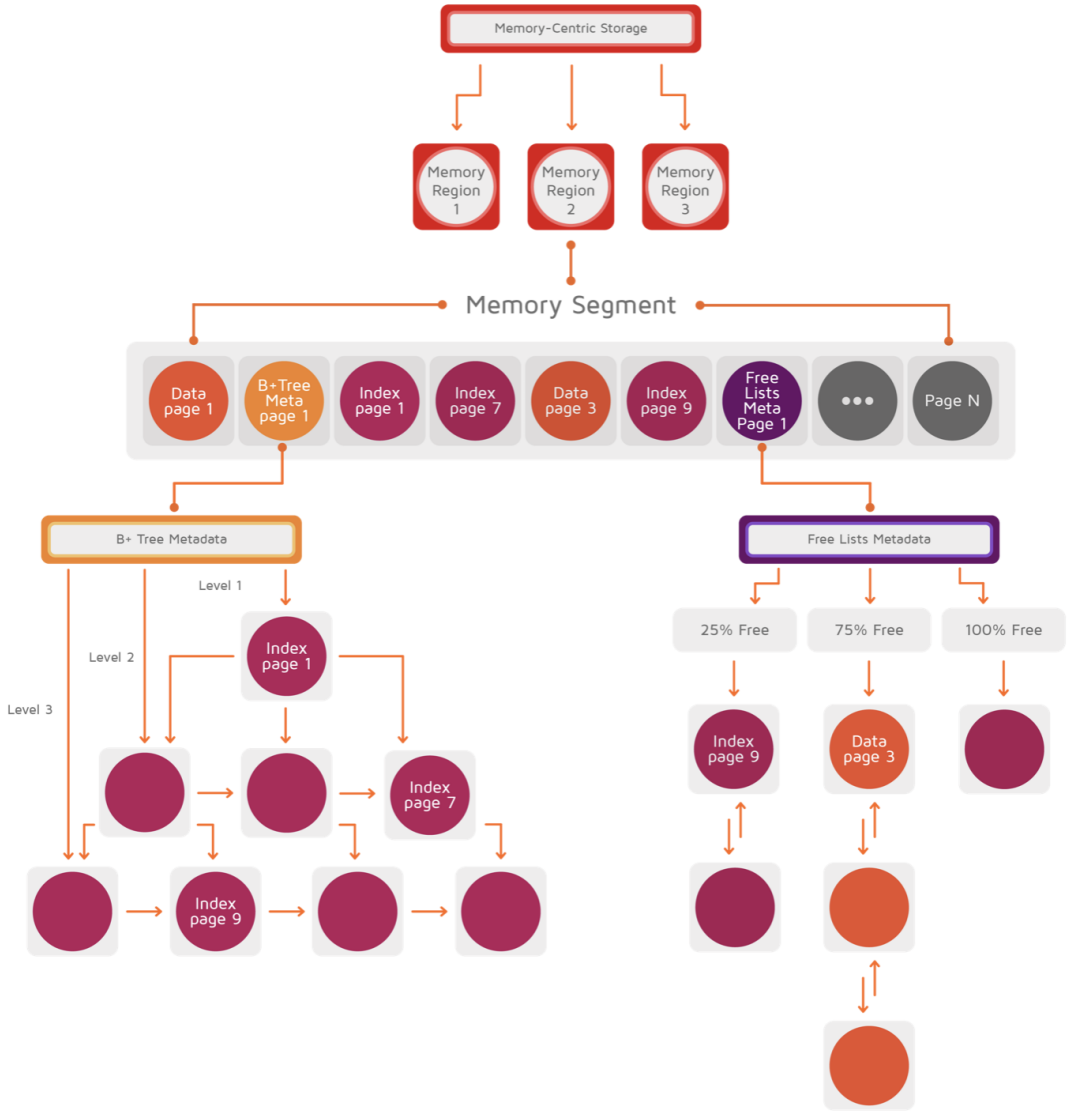

Figura 14: Organizzazione dell'architettura di storage multi-livello

Come già detto, il payload principale delle pagine è costituito dalle data entry degli oggetti memorizzati in cache lato applicazione. Una singola pagina mantiene in generale più coppie chiave-valore per usare più efficientemente la memoria ed evitare la frammentazione della stessa. Quando una nuova entry viene aggiunta alla cache, Ignite cerca una pagina ottimale, ossia quella che può contenere interamente la coppia chiave-valore. Può capitare, tuttavia, che la sua dimensione ecceda quella della pagina: in questo caso l'oggetto andrà ad occupare più di una pagina. Nel caso di aggiornamento di una entry esistente, che si espande oltre lo spazio disponibile della pagina che fino a quel momento la conteneva, Ignite cerca una nuova pagina che abbia abbastanza spazio per contenere interamente la coppia modificata e lì la sposta.

A seguito di queste modifiche e aggiornamenti, può capitare che nel tempo la pagina vada incontro ad una progressiva frammentazione della memoria, con un conseguente degrado delle performance. Per minimizzare questo effetto, Ignite usa un meccanismo automatico di *page compaction* attraverso cui attua una deframmentazione.

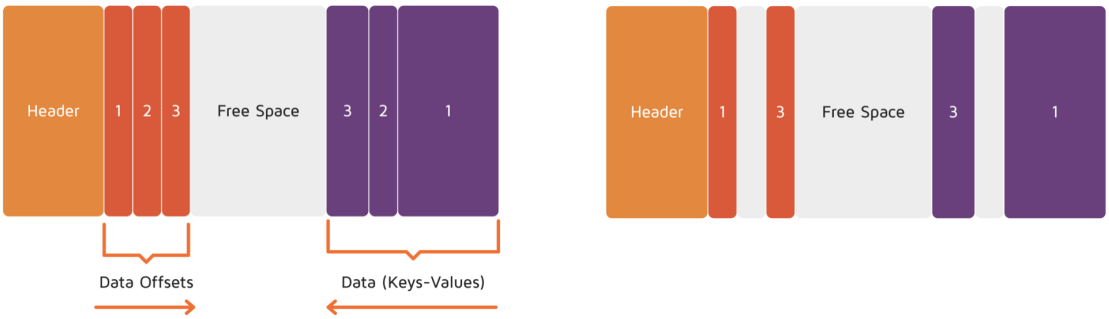

Figura 15: Layout di una pagina compatta (a sinistra) vs. layout di una pagina frammentata (a destra)

Come si può vedere dalla Figura 15 a sinistra, oltre ad avere un header che memorizza i metadati necessari all'uso interno, il layout di una pagina è strutturato in modo da memorizzare le coppie chiave-valore da destra verso sinistra e i rispettivi offset da sinistra verso destra. Infatti, le entry possono avere dimensioni differenti e gli offset (che invece hanno dimensione fissa) fungono da puntatori per accedere rapidamente ai dati di interesse.

Assumendo che ad un certo punto la data entry 2 della figura venga rimossa, si verrà a creare una situazione come quella in Figura 15 a destra, in cui lo spazio della pagina non risulta essere più continuo. Quando tutto lo spazio disponibile nella pagina viene richiesto, o quando una certa soglia di frammentazione viene raggiunta, si attiva quindi il processo di compattazione che porta il layout della pagina ad essere nuovamente continuo, similmente all'immagine di sinistra.

Quando la persistenza non è abilitata, Ignite memorizza di default le entry delle cache in memoria off-heap, allocando pagine man mano che i dati vengono aggiunti. Quando si raggiunge il limite di memoria della regione, l'utente deve decidere come gestire questa situazione configurando un'appropriata *eviction policy*, che determina quali entry debbano essere cancellate. Tale scelta si applica alla singola data region, che deve essere opportunamente configurata, come mostrato nel Capitolo 5.3. Di default, Ignite fa cominciare il processo quando il consumo di RAM per la regione raggiunge il 90%, ma si può specificare una soglia a piacere.

Scegliere con criterio le cache entry da rimuovere è fondamentale, dato che l'eliminazione delle entry in mezzo a una pagina causerebbe la frammentazione della stessa. Pertanto, Ignite applica uno degli algoritmi preconfigurati per selezionare la pagina più adatta alla eviction, da cui cancellare una per una tutte le entries. L'unica situazione in cui ciò non succede è quando una delle entry è bloccata da una transazione in corso e quindi viene mantenuta. A seconda delle situazioni, quindi, o l'intera pagina o una

sua ampia fetta viene svuotata e diventa pronta per essere riutilizzata, come mostrato in Figura 16.

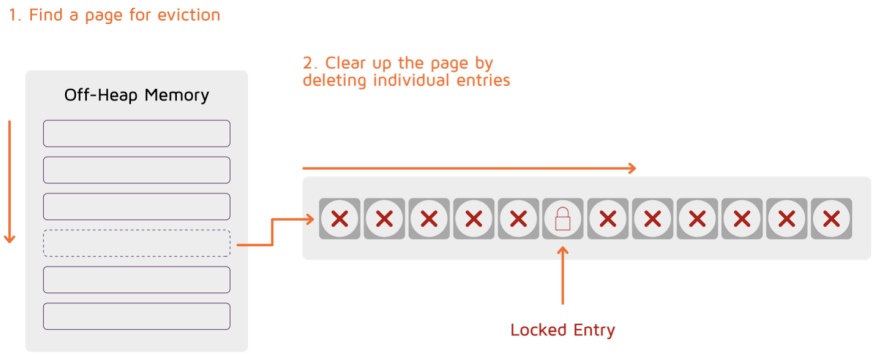

Figura 16: Processo di entry eviction per una cache off-heap senza persistenza abilitata

Ignite supporta due algoritmi per selezionare la pagina per l'eviction:

- **Random-LRU**: un array viene allocato per tenere traccia dei timestamp associati all'ultimo utilizzo di ciascuna pagina. Ogniqualvolta una pagina viene acceduta, il relativo timestamp viene aggiornato. Quando è necessario eliminare una pagina, l'algoritmo sceglie in modo casuale 5 indici dall'array ed elimina la pagina con il timestamp più vecchio. Se l'indice punta a una pagina senza dati, ne viene scelta un'altra.
- **Random-2-LRU**: in questo caso, per ciascuna pagina vengono memorizzati i due timestamp più recenti. Alla fine del processo di eviction, l'algoritmo sceglie in maniera casuale 5 indici dall'array e il timestamp più piccolo tra i due viene preso per essere comparato con i corrispondenti minimi delle altre quattro pagine scelte come candidati. Si tratta di una versione migliorata del precedente algoritmo, in grado di non andare ad aggiornare il timestamp per quei dati acceduti (recentemente) una sola volta, ma che in generale raramente vengono utilizzati.

Mentre il processo di *entry eviction* appena descritto si applica nel caso in cui Ignite operi in modalità *memory-only* (vedi Capitolo 2.2.2) o in modalità *external storage* (vedi Capitolo 2.2.4), o ancora quando sono configurate le *near cache* (vedi Capitolo 2.2.1), nel caso in cui sia attiva la modalità *native persistence* (vedi Capitolo 2.2.3) viene invece impiegato un processo simile, chiamato *page eviction* (detto anche *page replacement* o *page rotation*). La differenza sostanziale sta nel fatto che in questo caso si è meno preoccupati di perdere i dati, essendo tutti presenti anche su disco, e più focalizzati sul migliorare l'efficienza. In queste situazioni, infatti, la quantità di dati che Ignite memorizza sul disco è tipicamente maggiore rispetto alla capacità di memoria della cache off-heap, pertanto si rende necessario individuare una pagina da spostare dalla cache al disco per caricarne una nuova dal disco alla cache.

Il processo di sostituzione procede nel modo seguente: quando Ignite richiede una pagina, prima la cerca sulla memoria off-heap. Se non la trova, viene pre-caricata dal disco. Nel mentre, se la memoria off-heap è già piena, viene scelta anche un'altra pagina da memorizzare su disco e da sostituire.

In questo caso, gli algoritmi supportati sono tre:

- **Random-LRU**: si tratta dello stesso algoritmo impiegato anche nel caso della entry eviction. Non risulta essere particolarmente efficace nel trovare la prossima pagina da sostituire, quindi il suo utilizzo conviene quando si lavora con data region abbastanza grandi da contenere tutti i dati necessari, o comunque quando la rotazione è necessaria solo di rado.

- **Segmented-LRU**: è una variante di LRU, in cui la lista di pagine viene divisa in due segmenti, *probationary* (*i.e.* in prova) e *protected*, in ciascuno dei quali le pagine sono ordinate in base all'accesso, dal meno recente al più recente. Le nuove pagine vengono aggiunte in coda al primo segmento, mentre le pagine esistenti vengono rimosse dalla posizione in cui risiedono attualmente e aggiunte in coda al segmento protetto: le pagine di quest'ultimo segmento vengono quindi accedute almeno due volte. Il segmento protetto è finito, quindi la migrazione di una pagina dal segmento in prova al segmento protetto può forzare la migrazione della pagina acceduta meno di recente nel segmento protetto alla coda del segmento in prova (*i.e.* dove sta la pagina acceduta più di recente). Questo dà alla pagina un'altra possibilità di essere acceduta prima di essere sostituita. La pagina da sostituire viene presa dalla testa del segmento in prova, dove si trova la pagina acceduta meno di recente. A fronte di una richiesta di memoria aggiuntiva per memorizzare l'elenco delle pagine, il quale deve anche essere aggiornato ad ogni nuovo accesso, questo algoritmo ha una policy di selezione della pagina da sostituire quasi ottimale, ragion per cui può superare gli algoritmi Random-LRU e CLOCK quando si ha un alto tasso di rotazioni di pagina e di scansioni singole. Per lo stesso motivo, può risultare leggermente meno performante rispetto agli altri nel caso in cui non si verifichino sostituzioni di pagina.

- **CLOCK**: è l'algoritmo di default, adatto nella maggior parte delle situazioni, con un'efficienza nella policy di sostituzione che si posiziona tra Random-LRU e Segmented-LRU. Mantiene una lista circolare delle pagine in memoria, con il puntatore che tiene traccia dell'ultimo frame di pagina esaminato. Quando si verifica un page fault e non esistono frame vuoti, la hit flag della pagina viene ispezionata nel punto in cui si trova il puntatore della lista. Se la hit flag è 0, la nuova pagina viene messa al posto della pagina a cui punta il puntatore, che avanza di una posizione. In caso contrario, la hit flag viene cancellata, il puntatore viene incrementato e il processo viene ripetuto fino alla sostituzione di una pagina.

La sostituzione delle pagine potrebbe avere influenze negative sulle performance, con Ignite che elimina ripetutamente pagine dalla RAM che dopo poco sono di nuovo richieste. In questi casi è necessario rileggere i dati dal disco.

Per completezza di esposizione si precisa che, nel caso vengano utilizzate le *on-heap cache* (vedi Capitolo 2.2.1), oltre all'algoritmo LRU sopra descritto, possono essere scelti anche i seguenti algoritmi:

- **First In First Out** (**FIFO**): l'entry che si trova nella cache da più tempo è quella che viene eliminata per prima. Si differenzia da LRU per il fatto che viene ignorato l'ordine effettivo di accesso alle entry.
- **Sorted**: simile al precedente algoritmo, si differenzia per il fatto che l'utente può specificare un ordinamento alle entry configurando un comparatore.

A differenza degli altri casi, queste ultime due policy supportano anche la batch eviction, che permette di far partire il processo solamente quando la dimensione della cache eccede di `batchSize` la dimensione massima della cache, in aggiunta al comportamento classico che fa partire l'eviction non appena si supera il limite massimo della cache.

2.2.1 Cache

Più volte nel corso della trattazione si è fatto riferimento al concetto di cache. In questa sezione se ne approfondiscono i dettagli e le diverse strategie di utilizzo.

Una delle principali caratteristiche di Apache Ignite è infatti la capacità di fornire quella che in gergo viene chiamata *in-memory data grid*, vale a dire un layer che si va ad aggiungere tra l'applicazione e il data store già esistenti e che sfrutta la memoria RAM, nello specifico una cache, per memorizzare i dati acceduti più di frequente e quindi migliorare le performance del sistema.

Una siffatta architettura incarna il principio di *Cache-as-a-Service* (CaaS), dal momento che non è necessario implementare un'infrastruttura di cache locale ad ogni applicazione, le quali invece interagiranno con questa interfaccia distribuita comune. In questa modalità, la cache si estende su più server (che compongono la griglia) e viene mantenuta sincronizzata in ciascuno di essi.

Per i dettagli implementativi su come configurare una specifica cache fare riferimento al Capitolo 5.4.

Caching Strategy Sulla base di come viene usata la cache in un ambiente distribuito, possono essere definite tre strategie differenti:

- **Cache-aside**: in questo approccio, illustrato in Figura 17 della pagina seguente, è l'applicazione ad essere responsabile di leggere/scrivere dal/sul persistent store, visto che la cache non interagisce con il database. Prima di interrogare il DB, l'applicazione controlla se nella cache sono presenti i dati di cui ha bisogno e si occupa anche di aggiornare la cache dopo avere apportato modifiche al data store. Pur essendo molto veloce, questa strategia presenta alcune problematiche: innanzitutto, il codice applicativo può diventare molto complesso e duplicato se più applicazioni si interfacciano allo stesso data store. Inoltre, quando mancano dei dati in cache, l'applicazione deve interrogare il DB, aggiornare la cache e continuare

il suo processing e ciò si traduce in molteplici visite al data store se più applicazioni eseguono delle elaborazioni nello stesso momento.

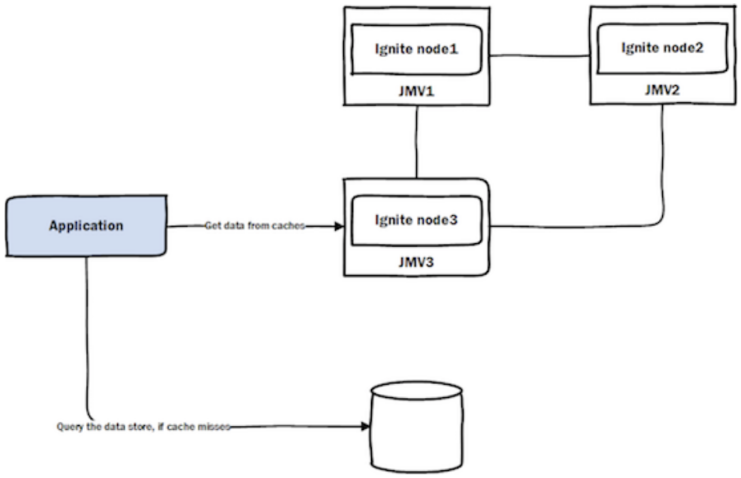

Figura 17: Cache-aside

- **Read-through** e **Write-through**: in questo caso (Figura 18), l'applicazione usa la cache come data store principale ed è quest'ultima che si occupa di leggere i dati dal DB in caso non siano disponibili in cache (read-through)[2] e di memorizzarli sul persistent store quando vengono aggiornati in cache (write-through). Rispetto al precedente approccio, semplifica notevolmente il codice applicativo, dato che è la cache, comune a tutte le applicazioni, che si interfaccia automaticamente al database.

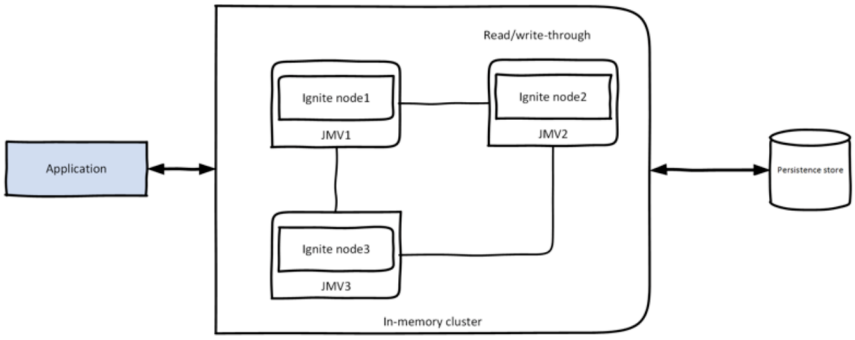

Figura 18: Read/Write-through

[2]Questo è vero solo per le operazioni di get eseguite tramite la key-value API (vedi Capitolo 2.4.1). Se si utilizza la SQL API, infatti, le query di SELECT non sono in grado di leggere dal database e hanno bisogno che tutti i dati necessari vengano pre-caricati in cache.

- **Write behind**: per ottenere performance migliori in scrittura è possibile adottare questo approccio (Figura 19), che prevede l'aggregazione degli aggiornamenti che vengono poi riversati sul disco in maniera asincrona tramite un'operazione bulk, che può avvenire su base temporale (specificando il tempo massimo in cui un certo dato può rimanere nella coda), in base alla dimensione della coda (specificando la dimensione massima che può raggiungere) o in entrambi i casi (in base a quale evento si verifica per primo). Di conseguenza, se un dato viene aggiornato sequenzialmente più volte, solo l'ultimo aggiornamento verrà effettivamente propagato al persistent store. Sebbene tutto ciò porti ad un incremento di performance, d'altra parte è più elevato il rischio di inconsistenza nel caso in cui un nodo fallisca.

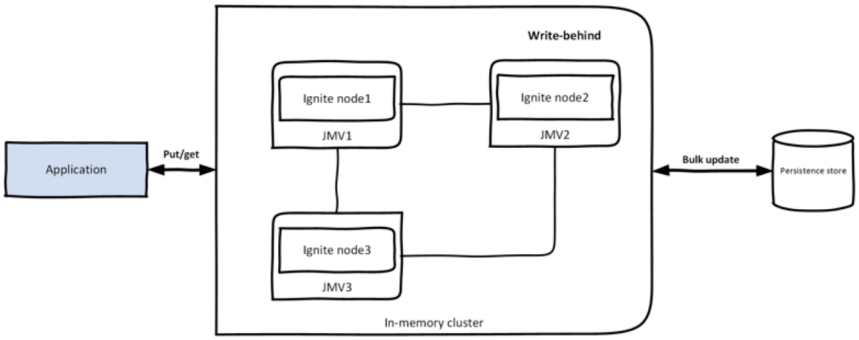

Figura 19: Write behind

On-heap cache vs. Off-heap cache Di default, la cache in Ignite viene creata off-heap, cioè al di fuori dello heap Java. Ciò porta diversi vantaggi, tra cui avere uno spazio di storage potenzialmente illimitato e non andare incontro all'overhead dovuto all'attività di garbage collection. Di contro, gli oggetti devono essere prima serializzati per poter essere memorizzati nella memoria off-heap, pertanto recuperarli da qui è più lento rispetto che recuperarli dallo heap Java, ma più veloce rispetto al disco.

All'occorrenza, Ignite offre anche la possibilità di definire la cache on-heap, utile per gli scenari caratterizzati da molte letture sui nodi server.

Cache group Ciascuna cache definita sul cluster porta con sé un certo overhead, dovuto al fatto che viene suddivisa in partizioni (vedi Capitolo 2.4.2) il cui stato deve essere tracciato da ogni nodo. Inoltre, se è attiva la modalità native-persistence (vedi Capitolo 2.2.3), ogni partizione corrisponde ad un file su disco da cui Ignite legge e scrive attivamente.

Quindi, all'aumentare delle cache e delle partizioni:

- più spazio sullo heap viene occupato per le partition map (ogni cache ne ha una)
- più tempo è necessario a un nuovo nodo per aggiungersi al cluster

- più tempo è necessario per eseguire il rebalancing (vedi Capitolo 2.4.5)
- peggiori sono le performance del checkpointing (vedi Capitolo 2.2.3)

Sebbene con un numero esiguo di cache (*i.e.* fino a qualche centinaia) questi problemi non si notino, l'impatto inizia ad essere non trascurabile quando si arriva ad avere qualche migliaia di cache. La soluzione al problema è rappresentato dai cosiddetti *cache group*, ovvero raggruppamenti logici di cache che condividono diverse strutture interne, come le partition map, riducendo l'utilizzo generale di memoria. Fare riferimento al Capitolo 5.5 per i dettagli implementativi.

Dal momento che i dati delle cache facenti parte dello stesso gruppo vengono memorizzati in partizioni condivise, è necessario aggiungere a ciascuna chiave l'ID univoco della cache specifica a cui il dato appartiene.

Si fa notare, infine, che l'uso dei cache group potrebbe talvolta portare a peggioramenti di performance per quanto riguarda le operazioni di lettura e i lookup sugli indici, considerato che tutti i dati e gli indici sono mescolati in strutture dati condivise che rendono necessario più tempo per interrogarle.

Near cache Una *near cache* è una cache locale che memorizza i dati più recenti o più frequentemente acceduti sul nodo in cui si trova, sia esso di tipo server o client (thick). Viene configurata on-heap per una ben specifica cache (regolare) e mantiene solamente i dati che si trovano lì. Il suo stato viene automaticamente aggiornato a seguito dei cambiamenti che si verificano sulla cache regolare a cui fa riferimento, da cui eredita anche la maggior parte delle configurazioni.

Quando da un client node si fa richiesta per una cache che non è stata configurata per usare una near cache, si può decidere di creare comunque una near cache per la cache in questione, dinamicamente. La near cache creata sarà operativa solo per il nodo in cui è stata creata e permetterà di incrementare le performance andando a memorizzare i dati direttamente client-side.

Partition Loss Policy Durante il ciclo di vita del cluster, può capitare che alcune partizioni vadano perdute a causa della failure dei rispettivi primary e backup node, arrivando ad una situazione di perdita parziale dei dati. Quando la topologia del cluster cambia, Ignite controlla se il cambiamento ha comportato una perdita di partizioni e, in base alla policy configurata e alle impostazioni di regolazione automatica della baseline topology, consente o proibisce ulteriori operazioni sulla cache.

Per le cache puramente in-memory, quando una partizione viene persa non è possibile recuperare i dati corrispondenti a meno che non vengano caricati nuovamente sul cluster. Per le cache persistenti invece, i dati non vengono fisicamente persi e quando i nodi falliti o disconnessi si riuniscono al cluster dopo un riavvio, i relativi dati vengono ricaricati da disco.

Le partition loss policy supportate, configurabili a livello di singola cache, sono le seguenti:

- `IGNORE`: la perdita di partizioni viene ignorata e qualora vengano richiesti i dati delle partizioni perse il cluster restituisce dei valori vuoti. Questa policy può essere usata solo per cluster in-memory.
- `READ_WRITE_SAFE`: ogni tentativo di lettura o scrittura di una partizione persa genera un'eccezione.
- `READ_ONLY_SAFE`: come sopra, ma anche per le partizioni disponibili non è possibile compiere operazioni di scrittura. La cache è quindi fruibile unicamente in modalità read-only.

Expiry Policy Considerato che lo scopo tipico di una cache è quello di memorizzare dati di breve durata, che hanno bisogno di essere aggiornati frequentemente, può avere senso in certi casi configurare una expiry policy affinché i dati rimangano memorizzati solo per un certo periodo di tempo, detto *time-to-live* (TTL). Una volta scaduti, i dati vengono automaticamente rimossi dalla cache. Il tempo che deve passare può essere contato dalla creazione, dall'ultimo acceso o dalla modifica del dato. Di default, il TTL è impostato in modalità eager e Ignite crea un thread apposito per eliminare i dati scaduti in background. Se questa modalità viene disabilitata, invece, i dati vengono rimossi solamente quando vengono richiesti da un'operazione della cache, dal thread che la esegue.

Nel caso in cui Ignite operi in modalità external storage, i dati scaduti vengono rimossi solo dalla memoria di Ignite, mentre rimangono intatti nel database esterno.

2.2.2 Modalità in-memory

Nel suo funzionamento di base, Apache Ignite opera in modalità puramente in-memory, caratteristica che lo distingue ad esempio dai tradizionali database relazionali che si affidano agli SSD o agli HDD per il data storage. L'utilizzo della memoria RAM sia per le computazioni che per la memorizzazione dei dati lo rende particolarmente adatto per quelle applicazioni che necessitano di un accesso e di un'analisi real-time sui dati.

I dettagli su come viene gestita la memoria centrale sono già stati trattati nelle sezioni precedenti.

2.2.3 Modalità native persistence

Quando questa modalità è attiva, Ignite memorizza sempre la totalità dei dati su disco, caricando in RAM quanti più dati riesce per potervi accedere ed elaborarli più velocemente. L'intento è quello di garantire in maniera ancora più forte la persistenza dei dati e la fault tolerance, di modo che nessun dato venga perso indipendentemente dal tipo di failure che si potrebbe verificare.

Ciascun nodo server è responsabile della persistenza di uno specifico sottoinsieme dei dati, che corrisponde alle partizioni ad esso assegnate, incluse eventualmente quelle di backup. Ciascuna partizione viene memorizzata sul disco in un file separato, che

mantiene lo stesso formato dei dati di quando si trovano in memoria. Oltre a ciò, anche gli indici e i metadati vengono memorizzati su disco.

La native persistence viene configurata a livello di singola data region. Per maggiori dettagli fare riferimento al Capitolo 5.6.

Persistent Storage Directory Per la persistenza dei dati, vengono generati tre differenti tipologie di file: le *cache page*, i *checkpointing marker* e i *WAL segment*.

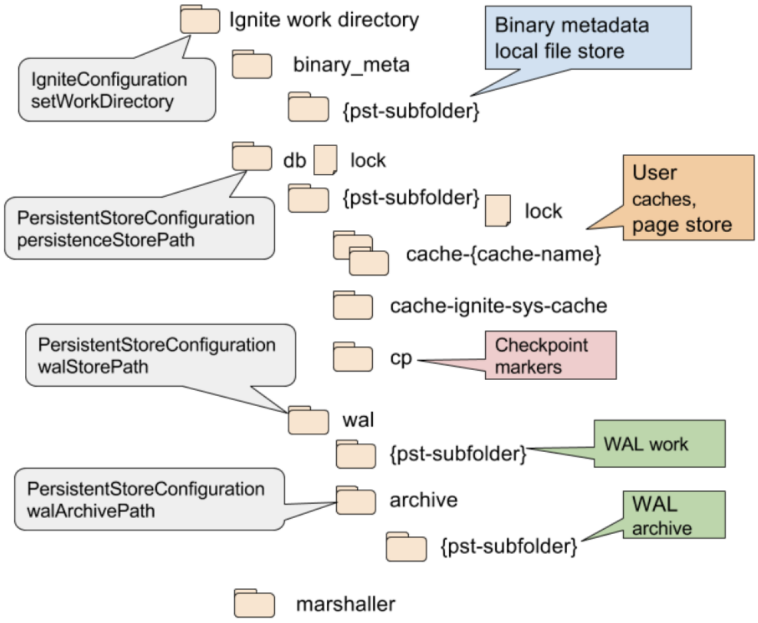

Figura 20: Struttura della cartelle generate in modalità native persistence

In Figura 20 è mostrata la gerarchia di cartelle generate automaticamente quando la persistenza è attiva. I file menzionati sopra vengono memorizzati all'interno della directory {`IGNITE_WORK_DIR`}`/db`. La sotto-directory {`pst-subfolder`}, che rappresenta uno specifico nodo del cluster, viene creata all'avvio e di default segue il pattern `nodeXX-UUID`, dove `node` è un prefisso costante, `XX` è un indice locale incrementale che identifica il nodo all'interno della directory corrente mentre `UUID` rappresenta l'ID univoco. Nello specifico, in {`IGNITE_WORK_DIR`}`/db/`{`pst-subfolder`} si trovano tutti i dati delle cache e gli indici (con una cartella differente per ciascuna cache creata), in {`IGNITE_WORK_DIR`}`/db/wal/`{`pst-subfolder`} sono contenuti i file WAL e in {`IGNITE_WORK_DIR`}`/db/wal/archive/`{`pst-subfolder`} i file di archivio WAL.

Nella selezione delle cartelle, vengono implementati due distinti meccanismi di lock. Il primo (situato in {`IGNITE_WORK_DIR`}`/db/`{`pst-subfolder`}`/lock`) viene usato per controllare che non ci siano nodi attivi e in esecuzione che stanno usando la stessa cartella, mentre il secondo (situato in {`IGNITE_WORK_DIR`}`/db/lock`) è tenuto solo per il tempo

necessario a creare una nuova cartella {`pst-subfolder`}, fronteggiando così l'inizializzazione concorrente di cartelle da parte di nodi che si avviano contemporaneamente.

Page Store Lo spazio su disco viene suddiviso in blocchi (pagine) di ugual misura. Ciascuna pagina ha un identificativo univoco che serve per localizzarla sul disco. Con il termine page store si fa riferimento allo storage che racchiude tutte le pagine associate ad una specifica cache. Come mostrato in Figura 21, ciascuna cache ha una propria directory (`cache-{cache-name}`) che rappresenta il suo page store, al cui interno si trovano i seguenti file:

- `part-0.bin`, `part-1.bin`, ..., `part-1023.bin` (`P1`, `P2`, `PN` in figura) sono le partizioni della cache, ciascuna contenente le relative pagine
- `index.bin` contiene i dati relativi agli indici delle partizioni
- `cache_data.dat` contiene informazioni di configurazione sui dati in cache

Quindi, ciascuna partizione corrisponde a un file e racchiude al suo interno più pagine.

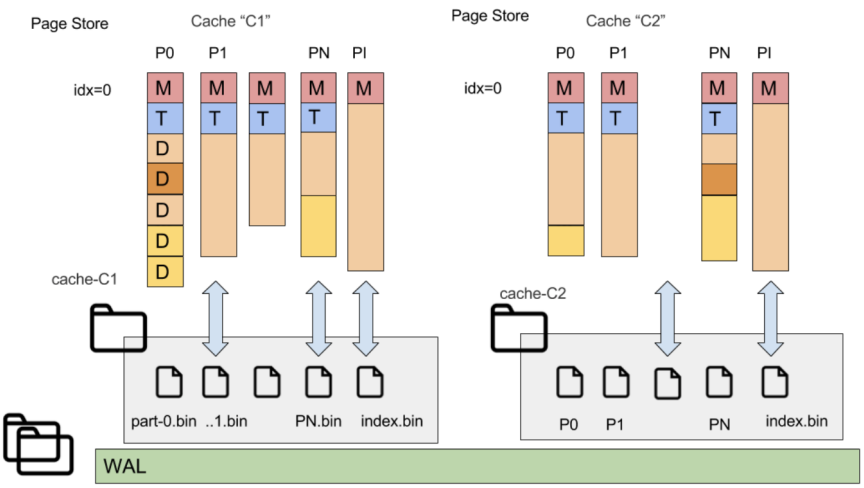

Figura 21: Suddivisione in pagine delle cache salvate su disco

Write-Ahead-Log (WAL) Ogni modifica che viene fatta ai dati in memoria RAM deve essere propagata anche su disco. Tradurre ciascun aggiornamento in-memory in un'operazione di write su disco risulta essere, tuttavia, un processo troppo lento. Per questo motivo, viene impiegata una tecnica chiamata *write-ahead logging*, che consiste semplicemente nell'accumulare in un file di log (WAL) tutte le modifiche avvenute alle pagine in RAM di un nodo prima di eseguire effettivamente l'aggiornamento sul disco. In questo modo, se si verifica un crash o un riavvio di un nodo, per ritornare all'ultima transazione

terminata con successo è sufficiente leggere e replicare le azioni nel WAL, sfruttando al contempo le informazioni salvate nel page store.

Lato pratico, non sarebbe fattibile fare il replay del WAL dall'inizio, dal momento che la capienza di un HDD è tipicamente minore della dimensione di un WAL intero. Si rende pertanto necessario un meccanismo che consenta di scartare le modifiche più vecchie contenute nel WAL, che prende il nome di *checkpointing*, illustrato nel paragrafo successivo.

Il WAL è composto da una serie di file, chiamati segmenti attivi, e da un archivio. I segmenti attivi vengono riempiti sequenzialmente e sovrascritti ciclicamente. Quando il primo segmento è pieno, il suo contenuto viene copiato nel WAL Archive; durante la copia del primo segmento, il secondo viene designato come segmento attivo ed accetta gli aggiornamenti che arrivano lato applicazione. Di default, possono esserci fino a 10 segmenti attivi. La Figura 22 illustra la struttura del WAL.

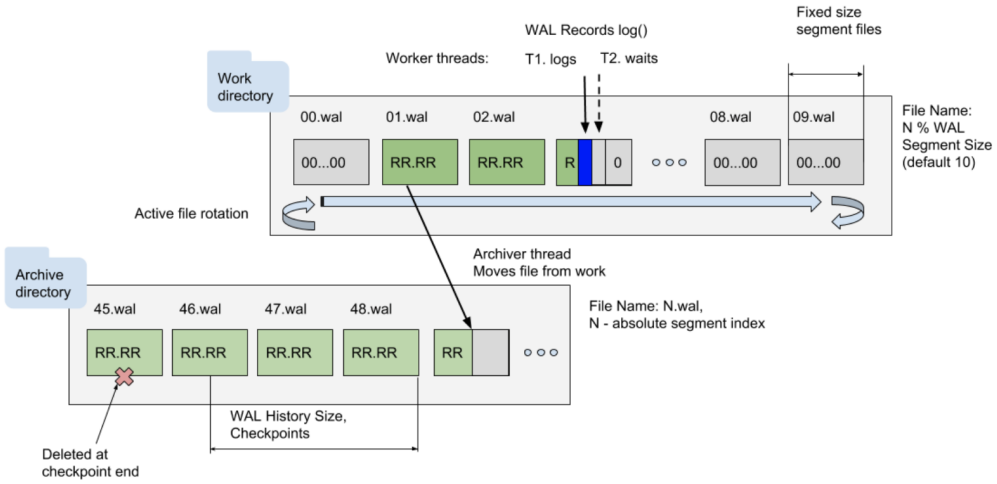

Figura 22: Meccanismo di rotazione dei segmenti

Il WAL Archive memorizza i segmenti che potrebbero servire per ripristinare un nodo dopo un crash. Il numero di segmenti che può ospitare dipende dalla sua dimensione, che di default è 4 volte la dimensione del buffer di checkpointing (vedi paragrafo successivo). Scegliere un valore appropriato per la dimensione del WAL Archive è importante per garantire buone performance, così come lo è trovare la giusta dimensione per i segmenti: segmenti troppo piccoli potrebbero risultare inefficienti in scenari con un alto carico di lavoro, dato che il meccanismo di logging passerebbe da un segmento all'altro troppo frequentemente e si tratta di un'operazione costosa. In certi casi, Ignite potrebbe riuscire a scrivere i dati sui segmenti WAL più velocemente di quanto i segmenti ci mettano ad essere copiati nell'archivio. Ciò può portare a un collo di bottiglia nell'I/O che può bloccare il funzionamento del nodo: disabilitare il WAL Archive può migliorare la situazione.

In altre situazioni, potrebbe invece convenire disabilitare del tutto il meccanismo di logging per avere prestazioni migliori, ad esempio durante il caricamento iniziale dei dati, ma bisogna prestare attenzione al fatto che se un nodo viene riavviato i relativi dati nel persistent storage vengono rimossi. Si tratta di un comportamento voluto, in quanto senza WAL non è possibile garantire la data consistency in caso di crash o riavvio di un nodo.

Esistono diverse modalità di WAL, che differiscono per il livello di consistenza garantita e quindi in maniera inversamente proporzionale per le performance:

- `FSYNC`: i cambiamenti vengono riportati su disco dopo ogni scrittura atomica o transazione. Nessun aggiornamento viene mai perso, nemmeno in caso di crash del sistema operativo.
- `LOG_ONLY`: è la modalità di default, in cui la sincronizzazione è a carico del sistema operativo. Gli aggiornamenti sono quindi preservati in caso di fallimento del processo, ma non del sistema operativo.
- `BACKGROUND`: in questo caso non viene fatto nulla al momento del commit, ma gli aggiornamenti vengono memorizzati in un buffer interno al nodo e inviati periodicamente al disco. In caso di crash del processo i dati potrebbero quindi venire persi.
- `NONE`: il WAL è disabilitato e i dati vengono aggiornati su disco solo in caso di gracefully shutdown del cluster.

Checkpointing Il checkpointing è il processo mediante il quale le *dirty page*, vale a dire le pagine che sono state aggiornate in memoria centrale e non ancora scritte nei rispettivi file di partizione (ma i cui aggiornamenti sono stati aggiunti al WAL), vengono copiate dalla RAM ai file di partizione su disco (Figura 23 della pagina seguente). Assieme al write-ahead logging, funge da meccanismo per garantire la durabilità dei dati e la possibilità di recuperarli in caso di failure dei nodi. Affinché il checkpointing possa iniziare, non occorre aspettare che una certa transazione in corso sia prima completata: il relativo commit verrà eseguito dopo che il checkpointing è terminato, oppure durante la sua esecuzione. Dopo aver completato con successo un checkpoint, è possibile eliminare i segmenti WAL creati prima di quel momento.

Può capitare che in parallelo al processo di scrittura delle pagine su disco, alcuni thread debbano aggiornare i dati presenti in una pagina in corso di scrittura o la cui scrittura è già stata programmata. In questi casi, viene adottata una tecnica di *copy-on-write* sfruttando una regione speciale, chiamata *checkpointing buffer*, nella quale Ignite crea una copia temporanea della pagina in oggetto nello stato in cui si trovava prima dell'aggiornamento concorrente.

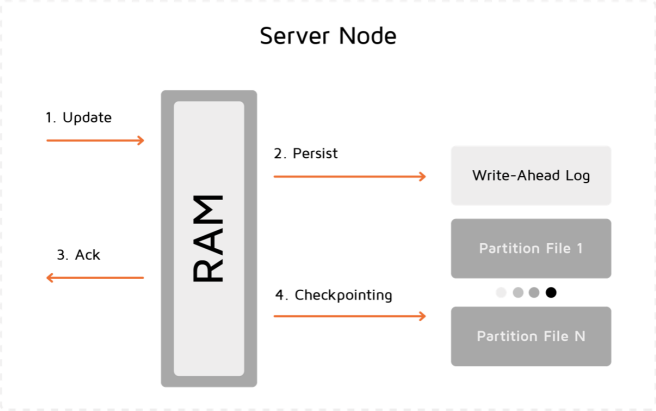

Figura 23: Processo di checkpointing

Tale meccanismo, schematizzato in Figura 24 della pagina seguente, procede nel modo seguente:

1. Viene acquisito un lock in scrittura per proteggere la pagina in oggetto da cambiamenti inconsistenti.
2. Viene creata una copia integrale della pagina nel checkpointing buffer.
3. I dati effettivi (a seguito dell'aggiornamento avvenuto in parallelo) vengono scritti in un segmento regolare.
4. La copia della pagina dal buffer viene scritta su disco.

Nel caso in cui una pagina non sia inizialmente coinvolta nel checkpointing, ma venga aggiornata in concomitanza con il processo di checkpointing successivamente, si procede in questo modo:

1. La pagina viene aggiornata direttamente in memoria bypassando il buffer di checkpointing.
2. I dati effettivi vengono memorizzati su disco nel corso del successivo checkpointing.

Il meccanismo appena descritto presenta dei limiti quando il throughput degli aggiornamenti ai dati è maggiore di quello del device fisico di storage, in quanto può portare all'overflow del buffer. Se si verifica questa situazione, Ignite blocca tutti gli aggiornamenti finché il checkpointing corrente non viene completato, di conseguenza in questo lasso di tempo le performance in scrittura scendono a zero. Lo stesso accade se la soglia sul numero massimo di dirty page viene raggiunta nuovamente mentre il checkpointing è ancora in corso. Ciò costringe Ignite a programmare un ulteriore checkpointing e bloccare tutte le operazioni di aggiornamento fino a quando il primo checkpointing non è terminato.

Una possibile soluzione a questi problemi può essere quella di abilitare l'algoritmo di throttling della scrittura delle pagine, che riduce le prestazioni delle operazioni di aggiornamento alla velocità del device di storage ogni volta che il buffer si riempie troppo velocemente o la percentuale di dirty page aumenta rapidamente.

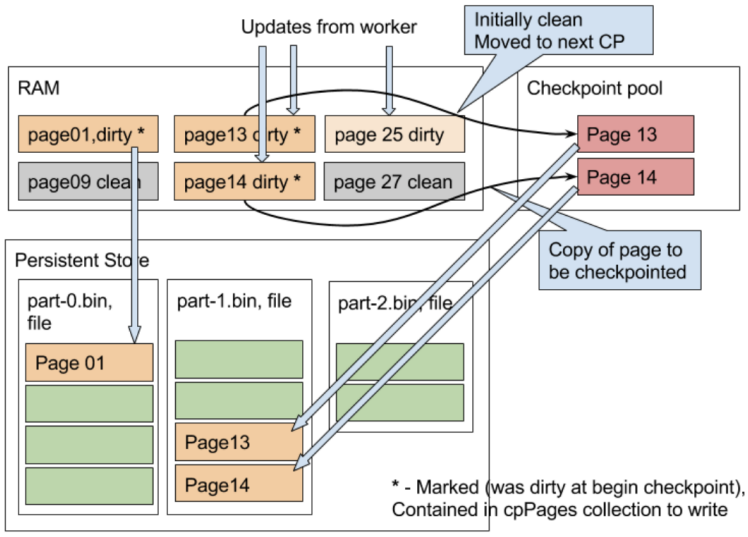

Figura 24: Processo di aggiornamento di una dirty page già programmata per il checkpointing

2.2.4 Modalità external storage

Un'altra tipica modalità di utilizzo consiste nell'usare Ignite come layer di caching al di sopra di un database esistente, che sia RDBMS o NoSQL, al fine di migliorarne le performance mettendo in campo il processing (e lo storage) in-memory. Tale modalità può essere altresì utile nel caso si voglia aggiungere il supporto a SQL a un database che non lo prevede, o in generale quando si preferisce memorizzare i dati su un supporto esterno piuttosto che usare la native persistence. Ignite fornisce un'integrazione nativa con Apache Cassandra[4], ma nel caso ci si voglia interfacciare ad altri database basta specificare la propria implementazione dell'interfaccia `CacheStore`.

In questa modalità, la cache opera secondo la strategia Read/Write-through, tuttavia questo significa che ogni operazione di inserimento e rimozione implichi una corrispondente richiesta al persistent store e di conseguenza la durata complessiva dell'aggiornamento potrebbe essere relativamente lunga, oltre che a comportare un carico elevato sullo storage in presenza di frequenti aggiornamenti. In questi casi è quindi possibile passare alla modalità Write behind, nella quale le operazioni di aggiornamento vengono eseguite in modo asincrono.

2.2.5 Swapping

Un'alternativa alle ultime due modalità illustrate è rappresentata dallo swapping. In questo caso, per evitare la saturazione della memoria RAM disponibile sui nodi, si fa affidamento allo spazio di swap su disco, direttamente gestito dal sistema operativo.

Ignite memorizza i dati nei *memory-mapped file* (MMF), il cui contenuto viene trasferito su disco dal sistema operativo in base al consumo di RAM corrente. Bisogna tenere in considerazione, però, che in questo scenario il tempo di accesso ai dati è più elevato, dal momento che lo spazio di swap si trova su disco. Inoltre, non vi è alcuna garanzia di durabilità, visto che i dati sono disponibili fintanto che il nodo è attivo: una volta che questo si spegne, i dati vengono persi. È pertanto sensato usare questa modalità come soluzione temporanea nel caso in cui si voglia scalare il cluster, per avere il tempo di aggiungere più nodi e ridistribuire i dati.

2.3 Architettura funzionale

Da un punto di vista funzionale, Ignite può essere visto come una raccolta di componenti in-memory indipendenti ma ben integrati, orientati a migliorare le prestazioni e la scalabilità di diversi tipi di applicazione. La Figura 25 riassume, in maniera non esaustiva, le funzionalità di base di Apache Ignite, la cui trattazione approfondita è demandata al Capitolo 3. Come si può notare, la loro organizzazione è modulare, e ciò si traduce nell'esistenza di un modulo (libreria) separato per ciascuna funzionalità. In questo modo, è possibile includere nel proprio progetto solamente le librerie effettivamente necessarie.

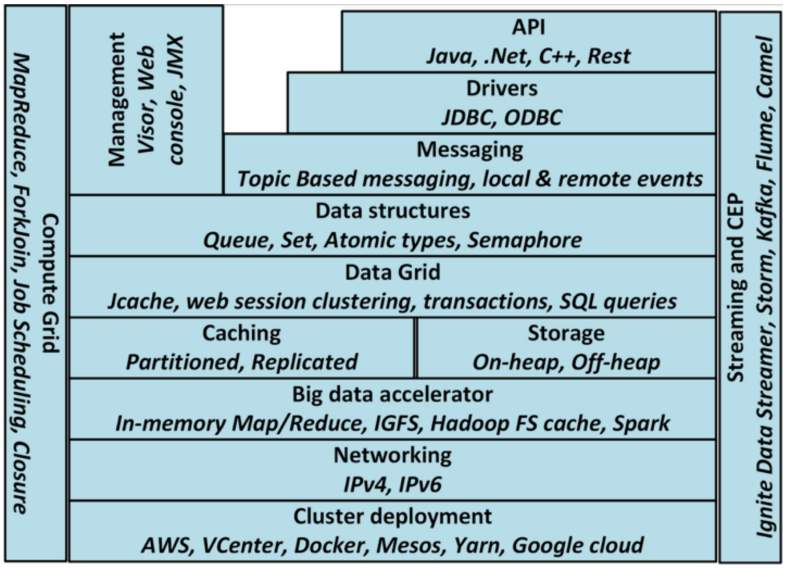

Figura 25: Architettura funzionale di Ignite

2.4 Data modeling

Apache Ignite si basa su un *data model* ben progettato che gli consente di migliorare le proprie performance, di utilizzare le risorse più efficientemente e in generale di facilitare l'utente a raggiungere i suoi obiettivi.

Per descriverlo, occorre prima capire come i dati vengono distribuiti all'interno di un cluster Ignite e le diverse modalità di accesso disponibili.

A **livello fisico**, i dati nel cluster sono organizzati in *data entry* memorizzate sotto forma di oggetti binari. L'intero dataset viene diviso in porzioni più piccole, chiamate *partizioni*, le quali vengono distribuite uniformemente tra tutti i nodi. La modalità con cui i dati vengono suddivisi in partizioni e come le partizioni sono collocate sui vari nodi è controllata da un'apposita *affinity function*.

A **livello logico**, i dati devono essere rappresentati in maniera tale da favorirne la gestione da parte dell'utente finale. È per questo che Ignite offre due distinte rappresentazioni logiche dei dati: la *key-value cache* e la *SQL table*. La loro differenza è puramente funzionale, dal momento che condividono la medesima struttura dati interna, a fronte di una diversa rappresentazione e modalità di accesso. Sono quindi equivalenti e possono essere usate in maniera interscambiabile, anche congiuntamente. Di seguito vengono brevemente descritte e messe a confronto.

2.4.1 Key-Value data model vs. SQL Table data model

Nella prima modalità di rappresentazione logica, i dati vengono visti come coppie chiave-valore che possono essere gestite mediante una specifica API. Il valore può essere un qualsiasi tipo di dato primitivo, come una stringa o un intero, o in generale un qualunque oggetto Java complesso. L'insieme di queste coppie dà origine ad un *key-value store*, che è la forma più semplice di data store del mondo NoSQL. Nello specifico, esso si basa sulla *JCache (JSR 107) specification*[5] che definisce una API standard per il caching in Java. Le operazioni che supporta sono molto semplici e riguardano perlopiù azioni di *put*, *get* e *delete*. In caso di inserimento di una chiave già presente in cache, il corrispondente valore viene sovrascritto con il nuovo valore fornito. Inoltre, tale data model sfrutta sempre l'accesso tramite chiave primaria, pertanto risulta essere performante e scalabile. Le principali operazioni supportate sono riassunte nella sezione Capitolo 5.7.

Nella seconda modalità, invece, i dati vengono rappresentati mediante tabelle che corrispondono alla tradizionale nozione di tabella degli RDBMS con qualche vincolo aggiuntivo, come il fatto che devono obbligatoriamente avere una chiave primaria. Questo fa sì che ciascuna tabella possa essere rappresentata anche come key-value cache, con la colonna relativa alla chiave primaria che funge da chiave e le restanti colonne che rappresentano i campi dell'oggetto, ovvero il valore. La Figura 26 della pagina seguente mette in evidenza l'equivalenza tra le due modalità appena descritte.

In conclusione, l'unica differenza tra le due rappresentazioni consiste nel modo in cui si accede ai dati. La key-value cache permette di lavorare agilmente tramite l'astrazione di oggetto usando linguaggi di programmazione supportati, mentre le tabelle SQL supportano la tradizionale sintassi SQL che può tornare utile ad esempio se si migra da un

database esistente. Differenziandosi solo a livello logico, è possibile creare dinamicamente sia cache chiave-valore che tabelle SQL e passare da una rappresentazione all'altra anche quando il cluster è già attivo e in esecuzione.

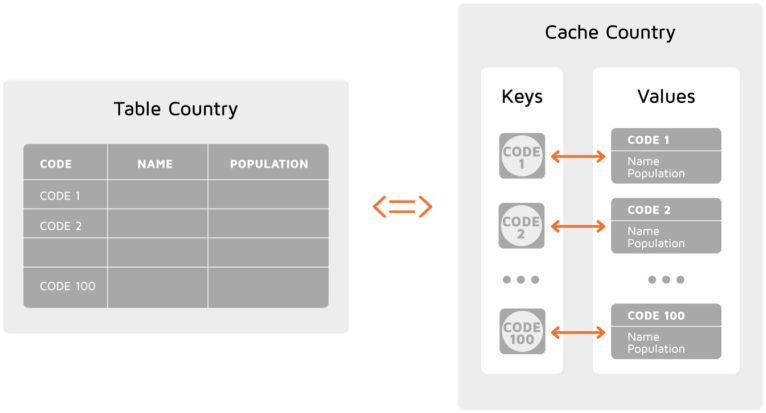

Figura 26: Rappresentazione dei dati mediante SQL Table (a sinistra) e Key-Value (a destra)

2.4.2 Data partitioning

Come già accennato precedentemente, i grandi dataset che devono essere memorizzati sul cluster devono essere sottoposti a un processo di suddivisione che li scomponga in parti più piccole più gestibili e distribuibili tra i diversi nodi. Tale operazione prende il nome di *data partitioning*.

Il problema della distribuzione dei dati tra i diversi nodi del cluster deve tenere conto di tre elementi essenziali:

1. **Algoritmo**: scegliere il giusto algoritmo permette ai nodi e alle applicazioni front-end che vi devono interfacciarsi di scoprire senza ambiguità in quale nodo o in quali nodi un certo oggetto (o chiave) si trova.

2. **Uniformità di distribuzione**: maggiore è l'uniformità di distribuzione dei dati sui nodi, maggiore sarà anche l'uniformità di workload sugli stessi.

3. **Minima interruzione**: se la topologia del cluster cambia a seguito del fallimento di un nodo, i cambiamenti nella distribuzione devono riguardare unicamente i dati che risiedevano sul nodo fallito. Allo stesso modo, se un nuovo nodo si aggiunge al cluster, nessun trasferimento di dati deve verificarsi tra i nodi che erano già parte della topologia.

Il partizionamento è controllato da una *affinity function*, la cui azione è duplice: per prima cosa determina il mapping tra le chiavi e le partizioni, che di default vengono identificate con un numero compreso tra 0 e 1023. Esse rappresentano al contempo dei

container per gli oggetti memorizzati in cache e delle unità di replicazione. Dopodiché, l'insieme delle partizioni viene distribuito tra i nodi server disponibili al momento, in modo tale che ciascuna chiave venga assegnata a un nodo specifico. Quando il numero di nodi del cluster cambia, a seguito di un inserimento o di una rimozione, le partizioni vengono automaticamente ridistribuite, mediante un processo di *rebalancing* (vedi Capitolo 2.4.5).

In Figura 27 è schematizzata ad alto livello l'azione della affinity function, che prende come argomento una *affinity key*, per la quale viene presa di default la chiave primaria, ma può essere scelto un qualsiasi campo degli oggetti memorizzati in cache.

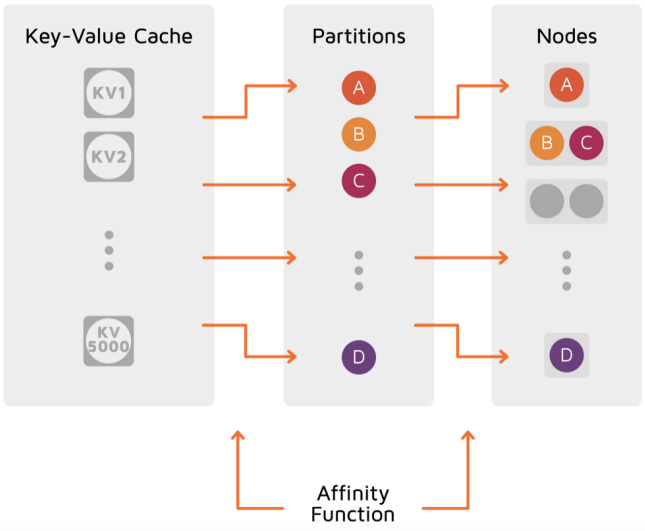

Figura 27: Azione della affinity function

Siccome il numero di partizioni in cui suddividere i dati viene stabilito in fase di configurazione e non cambia durante il ciclo di vita della cache, il mapping tra gli oggetti e le partizioni può essere eseguito tramite una semplice divisione modulo. Per quanto riguarda la distribuzione delle partizioni sui nodi, invece, Ignite fa uso dell'algoritmo di **Rendezvous Hashing**, descritto nel dettaglio nel Capitolo 4.1.

Per una distribuzione ottimale delle chiavi nelle partizioni e delle partizioni sui nodi è necessario che il numero di partizioni sia significativamente più grande del numero di nodi e che il numero di chiavi sia altrettanto più grande del numero di partizioni, come mostrato in Figura 28 della pagina seguente.

Tale algoritmo soddisfa il requisito di minima interruzione, poiché quando la topologia cambia, le partizioni vengono spostate solo da i nodi che hanno lasciato il cluster o verso i nodi che si sono aggiunti. Anche la proprietà di uniformità di distribuzione si può considerare rispettata con un buon grado di approssimazione, anche se essendo una soluzione randomizzata, è possibile che nel mapping partizione-nodo ci sia un bit di discrepanza, vale a dire che alcuni nodi potrebbero gestire un numero leggermente superiore di partizioni rispetto agli altri.

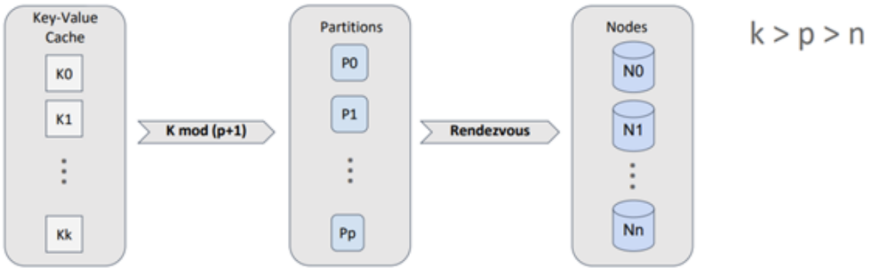

Figura 28: L'affinity function usa il modulo della divisione per il mapping chiavi-partizioni e l'algoritmo di Rendezvous Hashing per il mapping partizioni-nodi

In conclusione, il partitioning consente di migliorare notevolmente le performance, dal momento che garantisce che i dati vengano distribuiti equamente tra tutti i nodi, anche a seguito di un'aggiunta o una rimozione di nodi nel cluster, aiutando a raggiungere una scalabilità lineare virtualmente illimitata.

Inoltre, è possibile progettare il data model in modo tale che le data entry che vengono accedute/usate assieme siano anche memorizzate assieme, in un'unica partizione. Così facendo, quando si richiedono questi dati, solamente una piccola parte di partizioni deve essere visitata. Questa tecnica prende il nome di *affinity colocation*, ed è illustrata nel Capitolo 2.4.4.

2.4.3 Caching topology

In base a come la cache viene organizzata e gestita dal cluster, si possono venire a creare tre diverse topologie: **partitioned**, **replicated** e **local**. Tali modalità vengono associate individualmente e in maniera indipendente a ciascuna cache. Di seguito se ne analizzano i dettagli e i rispetti vantaggi e svantaggi.

Partitioned Si tratta della topologia predefinita e ha l'obiettivo di ottenere una *scalabilità* molto elevata. In questa modalità, infatti, Ignite partiziona in modo trasparente i dati memorizzati nella cache, distribuendo uniformemente il carico sull'intero cluster. Così facendo, la dimensione della cache e la potenza di elaborazione crescono in maniera lineare all'aumentare dei nodi disponibili. Ciascun nodo mantiene quindi una porzione dei dati, chiamata *primary partition*, e opzionalmente una o più *backup partition*, ossia delle copie di dati che sono memorizzati su altri nodi come partizioni primarie. L'uso delle copie di backup impedisce la perdita delle partizioni primarie nel caso in cui i nodi responsabili per esse (chiamati per estensione *primary node* per le chiavi memorizzate nelle suddette partizioni) diventino non disponibili.

Il vantaggio di questa topologia, mostrata in Figura 29 della pagina seguente, è che le operazioni di aggiornamento sono estremamente veloci, dato che è necessario aggiornare solo il primary node (e facoltativamente uno o più backup node) per ogni chiave. Per contro, le operazioni di lettura possono talvolta risultare più dispendiose perché solo

certi nodi hanno i dati necessari. Per queste ragioni, la modalità partizionata è ideale in presenza di dataset molto grandi e con aggiornamenti frequenti.

Per garantire una *high availability*, è pertanto necessario configurare una o più copie di backup della cache, che verranno collocate su nodi diversi dal primary. La seguente semplice formula permette di calcolare quante ne servono allo scopo:

$$number\ of\ backup\ copies = N - 1 \tag{1}$$

dove N rappresenta il numero totale di nodi nel cluster.

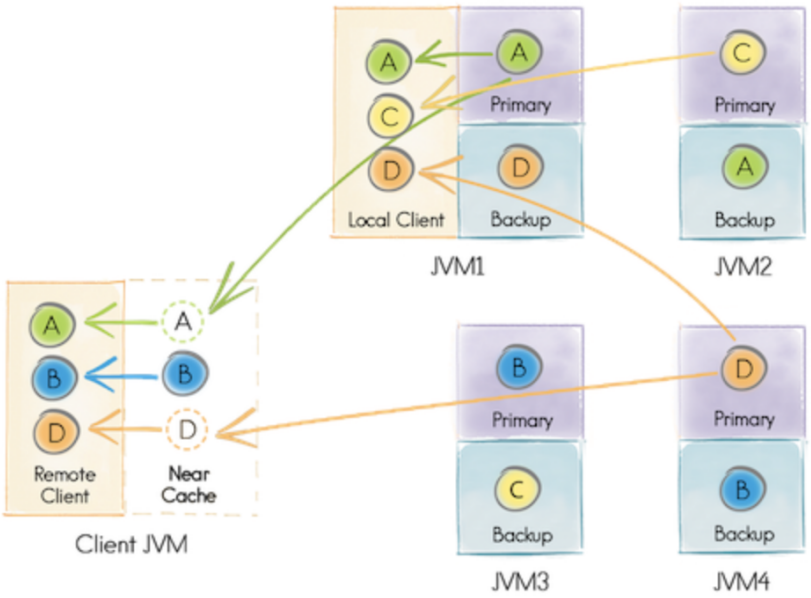

Figura 29: Partitioned caching topology

Replicated L'obiettivo di questo approccio è invece quello di garantire prestazioni molto elevate. I dati nella cache vengono replicati su tutti i nodi del cluster, i quali possono accedervi direttamente dalla propria memoria, senza aspettare. Ciò fornisce la massima velocità possibile per l'accesso in lettura, ma allo stesso tempo una tale ridondanza rende le operazioni di scrittura molto costose, dal momento che ogni aggiornamento ai dati deve essere propagato a tutti gli altri nodi, impattando sulle performance e sulla scalabilità generali. Detto in altri termini, la scalabilità della replicazione è inversamente proporzionale al numero di nodi, alla frequenza degli aggiornamenti e alla loro dimensione. Una cache replicata è quindi adatta in presenza di dataset piccoli e di aggiornamenti poco frequenti.

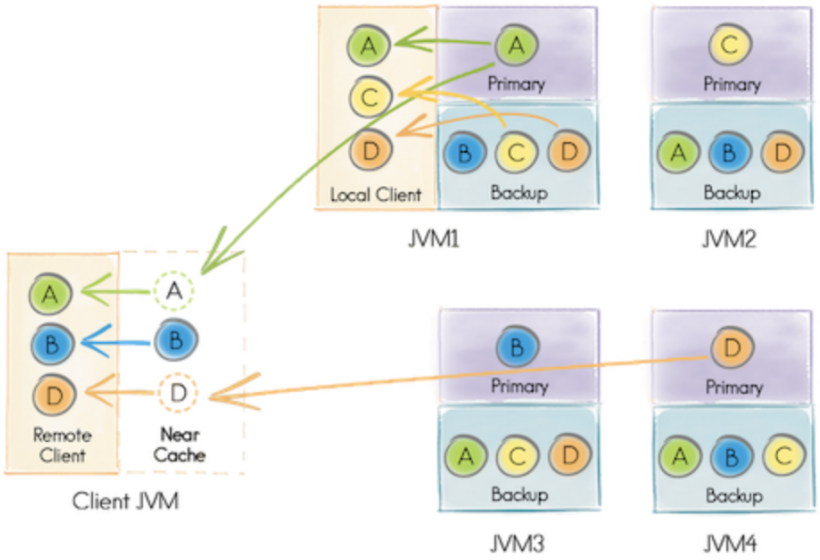

Figura 30: Replicated caching topology

Come si può notare dalla Figura 30, ogni nodo memorizza la totalità dei dati. Questo fa sì che la dimensione della cache replicata sia limitata dalla quantità di memoria disponibile nel nodo più piccolo.

Local Si tratta di una modalità ormai deprecata, che potrebbe essere rimossa nelle future release di Ignite. Con questo approccio non vi è alcuna distribuzione dei dati sugli altri nodi del cluster. Nonostante non vi sia alcuna logica di replicazione o partizionamento, l'ottenimento dei dati è molto veloce e poco costoso. Viene fornito un accesso a latenza zero ai dati utilizzati più recentemente e più frequentemente. Risulta utile per i dati di sola lettura o per quelli che scadono automaticamente dopo un certo intervallo di tempo e che devono essere ricaricati da disco.

2.4.4 Affinity colocation

Dal punto di vista delle performance, collocare sullo stesso nodo data entry multiple, che risultano essere in qualche modo correlate, è una scelta vincente. Tale correlazione può fare riferimento a una relazione di tipo padre-figlio tra gli oggetti o semplicemente al fatto che questi oggetti vengono spesso interrogati, e quindi acceduti, assieme.

Se gli oggetti sono distribuiti tra le varie partizioni attraverso la stessa affinity function, quelli con la stessa affinity key verranno a trovarsi nella stessa partizione e di conseguenza nello stesso nodo. In Ignite, tale approccio prende il nome di *affinity-based colocation* e permette di ridurre in maniera significativa la quantità di dati che vengono trasferiti tra i nodi per eseguire una computazione o una query.

Per capire meglio questo comportamento, si supponga di avere un data model composto da due cache/entità, `Order` e `OrderItem`, e che articoli multipli possono fare riferimento allo stesso ordine. Gli identificatori degli ordini e degli articoli sono indipendenti, ma ciascun articolo ha una chiave esterna che lo mette in relazione all'ordine a cui fa riferimento. Di default, l'affinity key è rappresentata dalla chiave primaria, pertanto la distribuzione degli ordini e degli articoli tra i nodi avviene in maniera del tutto indipendente gli uni dagli altri.

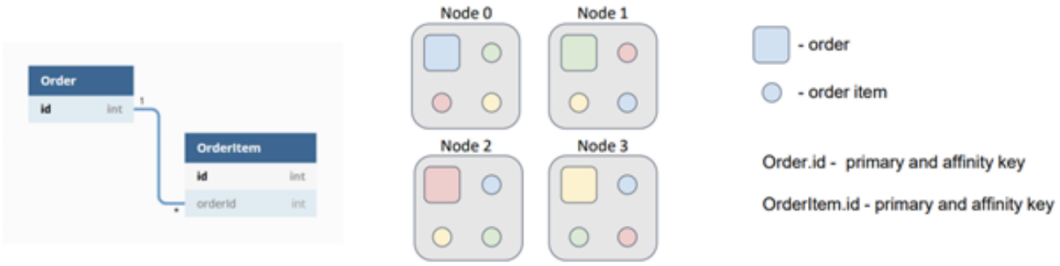

Figura 31: Disposizione sui nodi delle entità `Order` e `OrderItem` senza affinity colocation

Questa situazione è raffigurata in Figura 31, dove gli ordini sono rappresentati come quadrati e gli articoli corrispondenti come cerchi dello stesso colore dell'ordine. In questa situazione, un ipotetico task che coinvolge un certo ordine dovrà essere inviato presso il nodo in cui risiede, dopodiché per poter accedere agli articoli afferenti, che si trovano in generale su altri nodi, il task deve inviare dei sotto-task ai nodi interessati e collezionare i risultati. Tale interazione di rete risulta essere ridondante, ma può essere evitata facilmente facendo in modo che gli articoli relativi a un certo ordine vengano posti sullo stesso nodo in cui si trova quest'ultimo. Per farlo, basta designare la chiave esterna di `OrderItem` come affinity key, in modo tale che tale campo venga utilizzato per calcolare le partizioni e che gli articoli di un certo ordine vengano a trovarsi effettivamente sullo stesso nodo, come mostrato in Figura 32. Se entrambe le cache usano la stessa affinity function, i relativi dati di una e dell'altra verranno effettivamente co-locati sugli stessi nodi, evitando inutili comunicazioni di rete.

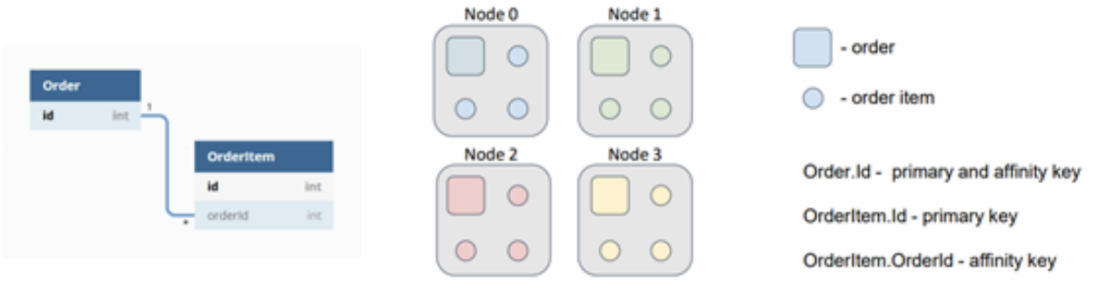

Figura 32: Disposizione sui nodi delle entità `Order` e `OrderItem` con affinity colocation

Per la guida su come definire nello specifico una affinity key personalizzata, fare riferimento al Capitolo 5.8.

È altresì possibile co-locare un task computazionale ai dati di cui necessita, come si avrà modo di spiegare nel Capitolo 3.1.

2.4.5 Data rebalancing

Il processo di *data rebalancing* garantisce che a seguito di cambiamenti nella baseline topology, (*i.e.* aggiunta o rimozione di nodi dal cluster) i dati rimangano equamente distribuiti andando a ricollocare le partizioni interessate in altri nodi. Da sottolineare il fatto che se un nodo, per cui non era stata prevista nessuna copia di backup, lascia permanentemente il cluster, le partizioni qui memorizzate verranno perse. In caso contrario, una delle copie di backup delle partizioni perse viene designata come nuova primary partition, mediante un processo chiamato **Partition Map Exchange** (PME), in seguito al quale il ribilanciamento ha inizio.

Il PME è un processo di condivisione delle informazioni riguardanti la distribuzione delle partizioni sul cluster, in modo tale che ciascun nodo sappia dove andare a trovare una specifica chiave. Questo meccanismo è richiesto in diverse situazioni, ad esempio in seguito al cambiamento della baseline topology, all'aggiunta o alla rimozione di una cache o alla creazione di un indice. Quando uno di questi eventi si verifica, il cluster aspetta che le transazioni correnti vengano completate prima di iniziare il PME, mentre tutte le nuove transazioni vengono posticipate alla fine di questo processo. Nello specifico, il nodo che funge per l'occasione da coordinatore del cluster richiede a tutti gli altri nodi le informazioni circa le partizioni che gestiscono. Una volta che il coordinatore ha ricevuto risposta da ciascun nodo, procede ad unire il tutto in una partition map completa, che viene inviata ai nodi. Il processo si conclude quando il coordinatore riceve la conferma di avvenuta ricezione da parte di tutti i nodi.

Nei cluster puramente in-memory, il rebalancing inizia di default immediatamente dopo l'uscita o l'aggiunta di un nodo, in maniera automatica. Nei cluster con la native persistence abilitata, invece, la baseline topology deve essere cambiata manualmente, a meno che non venga abilitata esplicitamente la regolazione automatica della baseline, descritta in precedenza.

Il ribilanciamento viene configurato a livello di singola cache e può essere impostato sia in modalità sincrona che in modalità asincrona. Nel primo caso, qualsiasi operazione sui dati della cache viene bloccata fino a quando il rebalancing non è terminato. Nel secondo caso, il processo di ribilanciamento viene portato avanti in maniera asincrona, cosicché la cache interessata sia comunque immediatamente disponibile per altre operazioni. Per come configurare la cache con una modalità di rebalancing specifica, fare riferimento al Capitolo 5.4.

2.4.6 Aderenza al teorema CAP

Quando si ha a che fare con un sistema distribuito è importante inquadrare quest'ultimo nell'ottica del ben noto **teorema CAP**, il quale afferma che in un'applicazione o in database distribuito non è possibile garantire simultaneamente l'aderenza a tre proprietà desiderabili, ma al più solamente a due di queste:

- **Consistency**: in un dato istante, ogni nodo del cluster deve poter vedere gli stessi dati, vale a dire la versione derivante dalla più recente operazione di scrittura terminata con successo.
- **Availability**: ogni nodo che non sia fallito deve sempre poter fornire una risposta in un lasso di tempo ragionevole a ciascuna richiesta che riceve.
- **Partition tolerance**: il sistema deve poter continuare a funzionare correttamente anche quando si verificano delle interruzioni di rete o quando un qualche nodo smette di funzionare.

In base ai requisiti specifici si possono presentare quindi le seguenti situazioni:

- **CA**: con questo approccio, si sacrifica la tolleranza alle partizioni di rete per garantire consistenza e disponibilità dei dati. È la situazione tipica dei database relazionali che svolgono i propri compiti tramite transazioni che rispettano le quattro proprietà **ACID** (**A**tomicity, **C**onsistency, **I**solation, **D**urability). Trattandosi di sistemi pensati prevalentemente per un uso a singolo server, presentano grossi problemi quando si cerca di distribuirli e quindi di scalarli.
- **CP**: viene sacrificata la disponibilità dei dati per garantirne la consistenza. Nel caso in cui uno o più nodi falliscano, si verifica una perdita di dati.
- **AP**: si sacrifica la consistenza dei dati per renderli sempre disponibili.

Dal momento che in un sistema distribuito le partizioni di rete sono la norma e che in quanto tale deve poter fornire un servizio che continui a funzionare nonostante le failure che potrebbero verificarsi, è chiaro che è necessario scegliere se prediligere l'*availability* piuttosto che la *consistency*.

I tradizionali data store NoSQL (*e.g.* Apache Cassandra) possono essere classificati come sistemi AP: essi non offrono supporto alle transazioni ACID in quanto l'accento è riposto sull'availability del sistema, che in questi casi è considerata essere più importante della consistency.

Sotto questo punto di vista, Apache Ignite non fa eccezione ed è classificabile come sistema AP. Tuttavia, come si avrà modo di spiegare in seguito, tra le varie funzionalità di Ignite c'è anche quella di offrire supporto alle transazioni ACID pur rimanendo tollerante alle partizioni, per questo motivo può essere classificato anche come sistema CP.

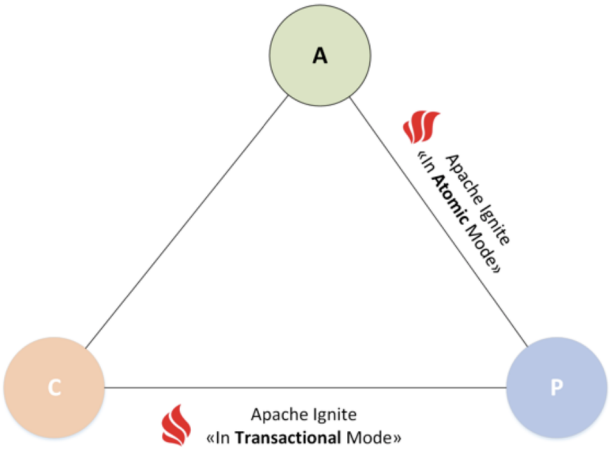

Figura 33: Posizionamento di Ignite rispetto al teorema CAP

Questo dualismo, illustrato in Figura 33, è reso possibile grazie alle due diverse modalità disponibili per le operazioni riguardanti la cache:

- **atomic mode**: si tratta della modalità attiva di default nella quale Ignite supporta operazioni atomiche multiple, ma eseguite una alla volta. Ciò garantisce performance migliori dal momento che non vengono impiegati lock transazionali. D'altro canto, poiché la data atomicity e la data consistency sono rispettate a livello di singola operazione, può capitare che le operazioni bulk (che non sono quindi eseguite in maniera transazionale) possano fallire parzialmente. Pertanto, viene data precedenza alla disponibilità del sistema (sistema AP).
- **transactional mode**: in questa modalità è possibile raggruppare in una transazione più operazioni, che verranno rese effettive solo in seguito al commit sulla cache. Se è stata configurata una copia di backup per la cache, Ignite userà il *2p commit protocol* (vedi Capitolo 4.2) per questa transazione. Da specificare che non si tratta di vere e proprie transazioni SQL, bensì di transazioni ACID-compliant. In questo scenario, Ignite acquisisce un lock pessimistico sui dati, quindi in caso di failure l'intera transazione non andrà a buon fine, mantenendo il data store in uno stato consistente. Viene quindi data la precedenza alla consistenza del sistema (sistema CP), a fronte di un maggior impatto sulle performance.

3 Feature

In questo capitolo si descrivono nel dettaglio le funzionalità offerte da Apache Ignite.

3.1 Distributed computing

In generale, il calcolo distribuito permette di suddividere un task di un'applicazione in più parti, di dislocarne l'esecuzione su più nodi di un cluster e di coordinare le loro attività in parallelo. Grazie a questo tipo di gestione si ha il vantaggio di poter eseguire le computazioni più velocemente, dato che è possibile sfruttare la potenza di calcolo non solo della macchina locale, ma anche delle restanti del cluster. Per servirsi al meglio della potenza di calcolo aggiuntiva è necessario che i task siano parallelizzabili.

Come si ha già avuto modo di sottolineare, Apache Ignite non è semplicemente un sistema di archiviazione dati distribuito, ma funge anche da *compute grid* distribuito, offrendo funzionalità differenti o migliorate rispetto agli approcci classici adottati da altri sistemi di computazione distribuita. Alcuni dei problemi principali di questi approcci sono la necessità di gestire la distribuzione di risorse e di task esplicitamente e l'assenza di un sistema di monitoraggio dell'avanzamento dei task e di riavvio in caso di errore.

Ignite fornisce delle API per gestire le computazioni distribuite e per sottomettere i singoli task ai diversi nodi del cluster in modo bilanciato e *fault tolerant*, con la possibilità di scegliere una specifica strategia di *job distribution* (vedi più avanti). Alternativamente, è possibile seguire il paradigma MapReduce, che permette la suddivisione automatica di più task.

Le caratteristiche principali del calcolo distribuito di Apache Ignite sono:

- **Data locality**: i task possono essere associati a specifici sottoinsiemi di dati.
- **Fork-join**: esiste un meccanismo standard per i fork-join che permette un trasferimento affidabile dei dati tra i nodi e il riavvio dei task in caso di errore.
- **Auto-deployment**: non è necessario effettuare il deployment del codice dell'applicazione/task su un server, ma è sufficiente connettersi a quest'ultimo e sottomettergli il task. Tutte le classi e le risorse vengono distribuite automaticamente (impostando la proprietà per il peer class loading, vedi Capitolo 5.9).
- **Checkpointing**: un task può salvare il suo stato in un checkpoint e in caso di errore ritornare allo stato salvato.
- **Failover** e **recovery**: vengono fornite funzionalità di ripristino per i task.
- **Scalability**: è possibile scalare fino a un numero arbitrario di nodi di computazione.
- **Asynchronous communication**: l'API supporta un modello di comunicazione asincrono che permette di gestire opportunamente la concorrenza.

- **Load balancing**: consente di bilanciare correttamente il carico nel cluster, tramite una serie di algoritmi.
- **Task session**: grazie al concetto di sessione, job separati, in esecuzione nello stesso task, possono vedersi e interagire tra loro.

Per i dettagli implementativi su come utilizzare l'API per la computazione distribuita, fare riferimento al Capitolo 5.9.

MapReduce Si tratta di un paradigma creato allo scopo di abilitare la computazione di grandi dataset parallelizzando il più possibile il lavoro. Inoltre, permette agli sviluppatori di concentrarsi sulla parte logica della computazione e non su quella di gestione dei dati. Il modello MapReduce consiste di 2 fasi:

- **Map**: su tutti i dati viene applicata una funzione di *map* che produce in output dei dati intermedi, in forma di coppie chiave-valore, da passare alla fase successiva. È la fase di computazione parallela vera e propria, dal momento che assume che il task che si vuole eseguire può essere suddiviso in più *job* indipendenti, che possono quindi essere eseguiti separatamente.
- **Reduce**: è la fase finale nella quale vengono aggregati i dati intermedi per chiave e viene applicata una funzione di *reduce* per ottenere come risultato zero, uno o più elementi, nel formato specificato.

Gli sviluppatori devono occuparsi unicamente dell'implementazione di queste due funzioni, senza doversi preoccupare del passaggio di dati che avviene tra i partecipanti alla computazione. In alcune implementazioni è possibile tuttavia aggiungere dei passaggi intermedi, allo scopo di aggregare e quindi ridurre di dimensione i dati prima della fase finale di reduce, in modo tale da minimizzare lo spostamento dei dati, con conseguente risparmio di tempo e miglioramento delle performance.

Ignite offre delle API per poter eseguire operazioni aderenti a questo paradigma, analizzate nel Capitolo 5.10. La computazione viene spartita tra i nodi in base alla specifica strategia di *load balancing* scelta e il risultato finale viene aggregato nel nodo che ha richiesto il task.

Se un nodo va in crash o non è più disponibile durante l'esecuzione di un task, tutti i job programmati per quel nodo vengono automaticamente assegnati ad un altro nodo. Questo è possibile grazie ai meccanismi di gestione di failover che Ignite utilizza. È comunque possibile gestire le eccezioni quando vengono lanciate e decidere di considerare il job come fallito.

Ignite crea una sessione distribuita per ciascun task, contenente le informazioni su quest'ultimo. Tale sessione è visibile dal task stesso e da tutti i job che genera, i quali possono così condividere attributi tra di loro. Questi attributi vengono assegnati ai job prima o durante l'esecuzione e diventano visibili agli altri job nello stesso ordine in cui sono stati inseriti.

Load Balancing Tutte le computazioni (job) provenienti dai task vengono automaticamente distribuite in maniera bilanciata tra tutti i nodi disponibili e specificati per la computazione. Gli algoritmi a disposizione per gestire la distribuzione dei job tra i nodi sono i seguenti:

- **Round-Robin**: è l'algoritmo applicato di default, distribuisce i job sequenzialmente su tutti i nodi specificati per un certo task. Supporta due modalità: "per-task" e "global". Nella prima modalità, viene scelto un nodo casuale all'inizio dell'esecuzione di ogni task come punto di partenza, dopodiché si itera sequenzialmente su tutti gli altri nodi: nel caso in cui un task venga suddiviso in tante parti quanti sono i nodi, questa modalità garantisce che tutti i nodi partecipino all'esecuzione dei job. Nella seconda modalità, una singola coda di nodi viene usata per tutti i task: a differenza del caso precedente, può capitare che se anche il numero di job di un task è pari al numero di nodi, alcuni job dello stesso task vengano eseguiti sullo stesso nodo se sono in esecuzione più task concorrentemente.
- **WeightedRandom**: sceglie un nodo in maniera casuale dalla lista di quelli disponibili. Si possono assegnare dei pesi ai nodi, in modo tale che quelli con un peso maggiore saranno assegnati proporzionalmente a più job. Di default il peso assegnato ai nodi è 10.

È possibile abilitare una funzionalità, chiamata *job stealing*, che permette ai nodi meno impegnati di "rubare" dei job a nodi in sovraccarico o bloccati nell'esecuzione. Ciò torna utile quando nella computazione ci sono job estremamente corti o veloci e altri che invece impiegano molto tempo e che farebbero aspettare in coda eventuali job successivi. Questa funzionalità consente a tutti gli effetti un *late* load balancing, dal momento che permette di riassegnare un job che era già stato precedentemente assegnato ad un altro nodo.

Fault Tolerance Come anticipato, Ignite possiede un meccanismo di tolleranza ai guasti integrato. Nel caso della computazione distribuita, la tolleranza si traduce nella capacità di gestire la scomparsa di nodi che stanno eseguendo dei job, i quali non vengono mai persi finché vi è almeno un nodo funzionante nel cluster. I job in questione vengono quindi trasferiti ad un altro nodo che possa occuparsene. Esistono diverse strategie di riassegnamento:

- **Always Failover**: impostata di default, cerca sempre di riassegnare il job fallito di un certo task procedendo prima con i nodi che ancora non hanno eseguito job di quel task e poi con quelli restanti.
- **Never Failover**: non effettua mai il riassegnamento del job fallito.
- **Job Stealing Failover**: questa implementazione deve essere usata solo se si vuole abilitare il job stealing.

Job Scheduling In Ignite, i job vengono mandati ad un pool di thread della JVM e vengono eseguiti in ordine casuale. Nonostante ciò, viene data la possibilità di usare logiche personalizzate per scegliere un ordinamento differente:

- **FIFO**: esegue i job nell'ordine in cui arrivano, con la possibilità di specificare il numero di job da eseguire in parallelo.
- **Priority**: i job vengono prima ordinati in base alla priorità e poi eseguiti. I job successivi vengono aggiunti alla coda e anche in questo caso si può specificare il numero massimo di job da eseguire in parallelo. È possibile incrementare la priorità dei job per evitare che rimangano in fondo alla coda per molto tempo.
- **Job Stealing**: abilita il job stealing dai nodi sovraccaricati a quelli liberi. È indicato nelle situazioni in cui c'è molta variazione nei tempi di esecuzione dei job.

Task-Data Colocation Le computazioni co-locate sono computazioni distribuite che eseguono il lavoro computazionale direttamente laddove i dati sono situati. I job vengono quindi inviati solo ai nodi interessati e solamente il risultato finale viene rimandato indietro. Questo modo di agire permette di minimizzare lo spostamento dei dati tra i nodi sulla rete e può migliorare i tempi di esecuzione.

Per i dettagli su come fare co-locare un task con i dati di cui ha bisogno fare riferimento al Capitolo 5.11.

3.2 SQL distribuito

Ignite offre supporto al linguaggio SQL, consentendo di interfacciarsi e di manipolare i dati in maniera molto più immediata per l'utilizzatore medio. Considerata la natura di un cluster, Ignite funge così da database SQL distribuito, fault-tolerant e scalabile orizzontalmente. Essendo conforme allo standard ANSI-99, vengono supportati tutti i comandi **DML** (Data Manipulation Language), con cui leggere, inserire, modificare ed eliminare i dati, oltre che a un sottoinsieme di comandi **DDL** (Data Definition Language) tramite i quali è possibile agire sullo schema stesso del database.

Le tabelle SQL presentano internamente la stessa struttura dati delle cache chiave-valore. Questa caratteristica permette di gestire le partizioni come meglio si crede e di sfruttare tecniche come l'affinity colocation per migliorare le performance.

Modalità di query Ci sono due modi principali per processare le richieste SQL:

- **in-memory MapReduce**: se la query è eseguita su cache in modalità partizionata, allora Ignite la esegue in maniera distribuita dividendola in più query di map e in una singola query di reduce finale. Le query di map, il cui numero dipende dalla quantità e dalla dimensione delle partizioni, vengono eseguite localmente presso i nodi in cui risiedono i dati coinvolti dall'interrogazione. Dopodiché, i loro risultati parziali vengono forniti al nodo che ha sottomesso la query iniziale e vengono qui integrati attraverso una query di reduce finale che fornirà il risultato complessivo.

Nulla vieta, tuttavia, di forzare l'esecuzione della query su un singolo nodo: ciò si tradurrà in un risultato parziale.

- **H2 SQL engine**: se, invece, la query è eseguita su cache replicate o locali, Ignite sa che tutti i dati sono reperibili su uno stesso nodo, perciò può lanciare la query richiesta in locale sfruttando un **H2 database engine**, un database gratuito scritto in Java che può essere integrato a livello di singolo nodo.

Schemi Uno schema rappresenta la configurazione logica di tutto o di parte di un database relazionale. Ignite offre diversi schemi e la possibilità di crearne di personalizzati, quelli abilitati di default sono i seguenti:

- **SYS**: contiene delle viste di sistema con delle informazioni sui nodi del cluster. In questo schema non è possibile creare tabelle.
- **PUBLIC**: viene usato di default quando nessuno schema è stato specificato.

Gli schemi personalizzati possono essere creati al momento della configurazione del cluster oppure quando viene creata una cache.

Indici Le query su un database MySQL possono essere ottimizzate applicando degli indici ai campi delle tabelle. Quando il database raggiunge una dimensione dei dati molto elevata, infatti, il tempo di risposta sulle query comincia a rallentare esponenzialmente. Ottimizzando gli indici sul DBMS si può ridurre il tempo di risposta di trenta volte, migliorando sensibilmente l'efficienza dell'esecuzione.

Ignite crea indici per ogni primary key e campo con affinity key in maniera automatica. Quando si definisce un indice su un campo, Ignite crea un indice composto dal campo indicizzato e dalla primary key della cache. In termini SQL, significa che l'indice sarà composto da due colonne, la colonna che si vuole indicizzare e la primary key (vedi Capitolo 5.12).

SQL API Oltre ai comandi di DDL, Ignite mette a disposizione delle API con cui manipolare i dati. Nello specifico, viene data la possibiltà di eseguire le query SQL direttamente sulla cache e vengono supportati anche i join distribuiti per fare interrogazioni su dati che si trovano su cache differenti.

Maggiori dettagli sono forniti nel Capitolo 5.13.

Join distribuiti Il modo in cui i dati sono distribuiti e organizzati (a livello di affinità) è da tenere a mente quando si cerca di eseguire un join su dati che sono sparpagliati su molteplici nodi. Mentre in modalità replicata i dati si possono trovare tutti in qualsiasi nodo, il problema sorge quando si lavora in modalita partizionata. Nelle precedenti versioni di Ignite, per eseguire un join tra più cache/tabelle in modalità partitzionata si era costretti a rendere i dati co-locati per ottenere risultati corretti. Attualmente, invece, è possibile scegliere se eseguire il join in modalità *colocated* oppure *non-colocated*.

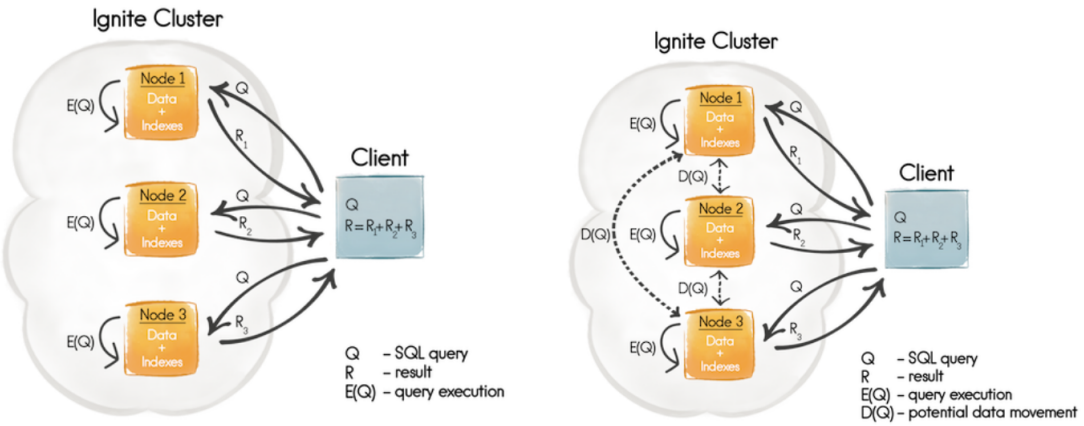

Figura 34: Colocated vs. non-colocated join

La differenza tra le due modalità è raffigurata in Figura 34. In entrambi i casi, la query `Q` viene inviata a tutti i nodi che contengono i dati che soddisfano la condizione specificata e viene quindi eseguita localmente su ciascuno di essi (`E(Q)`). Nel secondo caso, dal momento che i dati non sono co-locati, ogni nodo dovrà anche recuperare i dati mancanti che non sono presenti localmente, inviando delle richieste agli altri nodi (`D(Q)`). Tale richieste saranno di tipo unicast nel caso in cui il join venga fatto sulla chiave primaria o su una affinity key, in caso contrario saranno di tipo broadcast. Dopodiché, i singoli risultati $\texttt{R}_\texttt{i}$ delle query vengono rispediti al client che ha sottomesso la query iniziale, il quale provvede ad aggregarli formando il risultato finale `R`.

Si evince che i join co-locati sono più efficienti perché non necessitano di uno spostamento di dati tra nodi. Di default, Ignite tratta tutte le query di join come se quest'ultimi fossero co-locati. È quindi necessario fare attenzione alla reale organizzazione dei dati, perché se questi sono non co-locati si otterrà un risultato non corretto, a meno che non venga impostato un parametro specifico (vedi Capitolo 5.14).

Per aumentare le performance delle query con join, Ignite mette a disposizione un algoritmo di *hash join*, che risulta essere spesso più efficiente dei join ciclici innestati, sopratutto quando una delle due tabelle non è troppo piccola. Tuttavia, gli hash join possono essere utilizzati solo in presenza di equi-join, caratterizzati da un confronto di uguaglianza nel predicato di join.

Transazioni Le transazioni sono una sequenza di operazioni eseguite come se fossero una singola unità logica di lavoro. Le uniche due situazioni ammesse sono quelle in cui tutte le operazioni hanno successo o tutte falliscono: nel primo caso, le operazioni saranno effettivamente eseguite (*committed*), nel secondo caso, invece, se una o più operazioni falliscono, verranno annullate tutte (*rolled back*), quindi anche quelle che erano state eseguite correttamente verranno rimosse per non lasciare traccia di operazioni eseguite parzialmente, che potrebbero potenzialmente lasciare il database in uno stato inconsistente (vedi Capitolo 5.16).

Ottimizzazione delle prestazioni delle query Ci sono alcuni aspetti da tenere in considerazione per ottenere le migliori prestazioni possibili dalle query SQL in Ignite. Di seguito vengono elencati i principali:

- Scegliere con criterio dove usare gli indici: la loro presenza occupa infatti memoria (on-heap/off-heap). Inoltre, durante un update è necessario aggiornare separatamente ciascun indice coinvolto e nel caso di un update di dimensioni molto grandi, ciò può far degradare considerevolmente le performance.
- È consigliato indicizzare solamente i campi che verranno effettivamente usati nelle clausole `WHERE`.
- È consigliato prediligere l'utilizzo di join co-locati, che sfruttano quindi l'affinity colocation, piuttosto che quelli non co-locati. Quest'ultimi, infatti, porteranno a una maggior quantità di comunicazioni tra nodi in termini di dati e di messaggi per portare a termine le query.
- Prestare attenzione alle query che utilizzano gli operatori `OR`, dal momento che potrebbero fare un utilizzo degli indici che non ci si aspetta. Per esempio, nella query:

```
SELECT name FROM Person WHERE sex = 'M' AND (age = 20 OR age = 30)
```

  l'indice di `age` non verrà utilizzato anche se è una condizione più selettiva. Per ovviare al problema è necessario usare il comando `UNION ALL` nel modo seguente:

```
SELECT name FROM Person WHERE sex = 'M' AND age = 20
UNION ALL
SELECT name FROM Person WHERE sex = 'M' AND age = 30
```

3.3 Continuous query

Con il termine *continuous query* si fa riferimento a una query che viene applicata ad un flusso di dati continuo. Quando una continuous query viene fatta partire, Ignite osserva il cambiamento dei dati che avviene in una cache generando delle notifiche ogni volta che avviene una variazione nei dati che passano il filtro della query.

Considerata la loro natura, questo genere di query è particolarmente indicato per le notifiche di eventi, utilizzando un apposito *local listener* in esecuzione sul nodo che ha inizializzato la query (vedi il Capitolo 5.17 per maggiori dettagli). Quando una cache viene modificata da un'operazione di DML, un evento viene inviato al local listener della continuous query in modo tale che l'applicazione possa reagire nella maniera desiderata.

Si può anche specificare una query iniziale che viene eseguita prima che la continuous query venga registrata nel cluster e prima di iniziare a ricevere degli aggiornamenti da essa. Inoltre, è possibile specificare dei *remote filter* che permettono di restringere il range di record osservati durante gli aggiornamenti. Questo tipo di filtro viene eseguito per ogni chiave che viene aggiornata e valuta se l'update debba essere propagato al local listener della query. Una sua caratteristica importante è che può essere usato come

remote listener per gli eventi di update: questo è possibile dal momento che, per ragioni di ridondanza, il filtro viene eseguito sia sulla versione primaria che su quella di backup della chiave.

Considerato che di base le continuous query mandano l'intero oggetto aggiornato al local listener, potrebbe essere utile stabilire una funzione per specificare il sottoinsieme di campi di interesse come risultato di un ascolto remoto. Tale funzione, che prende il nome di *remote transformer*, viene eseguita sui nodi remoti per ogni oggetto che viene aggiornato e restituisce solo il risultato della trasformazione.

Le continuous query seguono la semantica *exactly-once* per la segnalazione di eventi al local listener. Questo è possibile grazie al fatto che sia il primary che i backup node mantengono una coda degli aggiornamenti contenente gli eventi che sono stati processati dalle continuous query lato server ma che devono ancora essere consegnate ai client. In più, Ignite gestisce un contatore speciale per ogni partizione che permette di non inviare notifiche duplicate.

3.4 Transazioni distribuite

Come già spiegato nel Capitolo 2.4.6, Ignite di default opera in modalità atomica, di conseguenza qualsiasi operazione bulk viene in realtà eseguita come sequenza di singole operazioni che possono venire intervallate, e quindi interrotte, da altre operazioni che potrebbero essere eseguite nello stesso istante, portando al fallimento di quella iniziale.

Per far fronte a questo, viene appositamente messa a disposizione una modalità transazionale (vedi Capitolo 5.18) che consente di raggruppare più operazioni che dal punto di vista logico devono essere eseguite in blocco, senza alcuna interruzione. Ciò significa che non sono consentite esecuzioni parziali: ciascuna operazione in oggetto ha successo oppure falliscono tutte.

Modalità concorrenti e livelli di isolamento Quando una cache è impostata in modalità transazionale, è necessario specificare un'opportuna modalità concorrente (*concurrency mode*), che determina quando una lock sui dati viene effettuato, ad esempio nel momento stesso di accesso al dato oppure nella fase di preparazione (*i.e.* subito prima dell'attuazione di un commit). Effettuare una lock previene qualsiasi tipo di accesso concorrente ad un oggetto. Indipendentemente dalla concurrency mode selezionata, ci sarà sempre, prima di un commit, un momento nel quale per tutti i dati compresi nella transazione è stata acquisita la lock.

Esistono due modalità concorrenti: **pessimistica** e **ottimistica**. Nella prima modalità, le transazioni acquisiscono la lock sui dati già nel momento della prima lettura (dipendentemente dal livello di isolamento scelto, vedi dopo) e la tengono fino al commit o al rollback. In questa modalità le lock vengono prese prima sui primary node e poi vengono propagate ai backup node. Nelle seconda modalità, invece, le lock non vengono mai ottenute finché non viene eseguito il commit. Le lock vengono prese sui primary node nella prima fase del two-phase commit protocol e poi propagate ai backup node.

Il livello di isolamento va a definire in che modo le transazioni distribuite vedono e gestiscono le operazioni sulle stesse chiavi. I livelli previsti sono tre e il loro compor-

tamento cambia in base alla modalità concorrente scelta. Di seguito si analizzano le caratteristiche e le differenze di comportamento di ogni possibile combinazione:

- Con concurrency mode **pessimistica**:
  - `READ_COMMITTED`: i dati vengono letti senza lock e non vengono mai memorizzati sulla cache nella transazione. È possibile specificare che i dati possano essere letti anche dai nodi di backup. Siccome può capitare che una transazione concorrente vada a modificare i dati che si stanno usando per la transazione, vengono adottate le cosiddette *non-repeatable read* e le lock vengono ottenute alla prima scrittura. Questo significa che un record che è stato precedentemente letto durante la transazione potrebbe avere un valore diverso nel momento del commit o di un'altra lettura, ma in tal caso non viene lanciata nessuna eccezione.
  - `REPEATABLE_READ`: le lock vengono prese al primo accesso di lettura o scrittura al dato. Tutti gli accessi successivi allo stesso dato sono locali e restituiscono l'ultimo valore letto o aggiornato nella transazione. Questo vuol dire che non ci possono essere transazioni concorrenti che fanno cambiamenti ai dati già acceduti, pertanto in questo caso sono consentite le *repeatable read*.
  - `SERIALIZABLE`: questa combinazione funziona allo stesso modo della precedente.
- Con concurrency mode **ottimistica**:
  - `READ_COMMITTED`: i cambiamenti che dovrebbero essere applicati alla cache sono memorizzati nel nodo d'origine e poi applicati al momento del commit. Come nel caso pessimistico, i dati transazionali sono letti (eventualmente anche dai nodi di backup) senza prendere lock e mai memorizzati sulla cache nella transazione. Non viene mai fatto un controllo sui dati per vedere se sono stati modificati dalla prima lettura e non viene lanciata nessuna eccezione nel caso.
  - `REPEATABLE_READ`: le transazioni funzionano in modo simile alla combinazione precedente con una sola differenza: i valori letti vengono memorizzati sul nodo d'origine e tutte le letture successive sono assicurate essere locali.
  - `SERIALIZABLE`: alla prima lettura di un dato viene associata una rispettiva versione. La transazione fallisce nella fase di commit se viene rilevato che almeno una delle entry utilizzate nella transazione è stata modificata (e ha quindi una differente versione), lanciando un'eccezione ed eseguendo il rollback degli eventuali cambiamenti effettuati.

Per i dettagli implementativi fare riferimento al Capitolo 5.19.

Read Consistency Per avere una consistency completa delle letture nella modalità pessimistica è necessario prendere le lock anche per questo tipo di operazioni. Ciò

è ottenibile utilizzando solamente `REPEATABLE_READ` o `SERIALIZABLE` come livelli di isolamento.

Queando si usa la concurrency mode ottimistica, invece, lo stesso comportamento può essere ottenuto con il livello `SERIALIZABLE`, che grazie al meccanismo delle versioni impedisce potenziali conflitti tra le letture.

Se non vengono usate le combinazioni appena riportate, è possibile che le transazioni si ritrovino in uno stato parziale: se una transazione aggiorna le entry x e y, un'altra transazione potrebbe vedere il nuovo valore per x e il vecchio per y.

Deadlock Un insieme di processi si trova in deadlock (*i.e.* in stallo) se ciascuno di essi è in attesa di un evento che solo un altro processo dell'insieme può provocare. Tipicamente, l'evento atteso è proprio il rilascio di risorse senza le quali i processi non possono continuare la loro esecuzione.

Una regola importante per lavorare con le transazioni distribuite è quella di prendere le lock delle chiavi partecipanti ad una transazione nello stesso ordine, altrimenti si rischia lo stato di deadlock. Ignite non dispone di veri e propri meccanismi in grado di evitare queste situazioni, ma fornisce funzionalità che permettono di effettuare il debug.

È possibile evitare completamente le deadlock utilizzando la modalità ottimistica e il livello di isolamento `SERIALIZABLE` dato che le lock non vengono prese sequenzialmente. In questa configurazione, le chiavi possono essere accedute in qualsiasi ordine poiché le lock sono prese in parallelo con un controllo aggiuntivo delle versioni che permette ad Ignite di evitare la deadlock.

In generale, se ci si ritrova in un ambiente molto concorrente, il locking ottimistico può portare ad avere un alto tasso di fallimento nelle transazioni, d'altro canto, in un ambiente caratterizzato da poche situazioni di contesa, può portare ad un miglioramento delle performance per transazioni molto grandi, siccome il numero di comunicazioni nella rete dipende solo dal numero di nodi su cui la transazione si appoggia e non dal numero di chiavi coinvolte nella transazione stessa.

3.5 Real time streaming

Apache Ignite fornisce nativamente dei *data streamer* per lo streaming di grandi quantità di dati verso il cluster, progettati in maniera tale da inoltrare flussi real-time senza fine verso le cache. Come mostrato in Figura 35 della pagina seguente, questa funzionalità è necessaria negli scenari in cui sono presenti diverse fonti che producono grandi moli di dati in modo continuo, che hanno bisogno di essere integrati ed elaborati simultaneamente direttamente sul cluster Ignite, in modo distribuito. In seguito all'elaborazione, i dati possono essere esportati in altri sistemi per la visualizzazione o per prendere decisioni di qualunque tipo.

Ignite fornisce diversi tipi di data streamer, ognuno con le proprie peculiarità. In generale, esso viene associato ad una specifica cache e i dati che riceve vengono automaticamente partizionati e distribuiti tra i nodi del cluster, in parallelo. Sempre in maniera concorrente, è possibile processare e interrogare i dati che man mano arrivano:

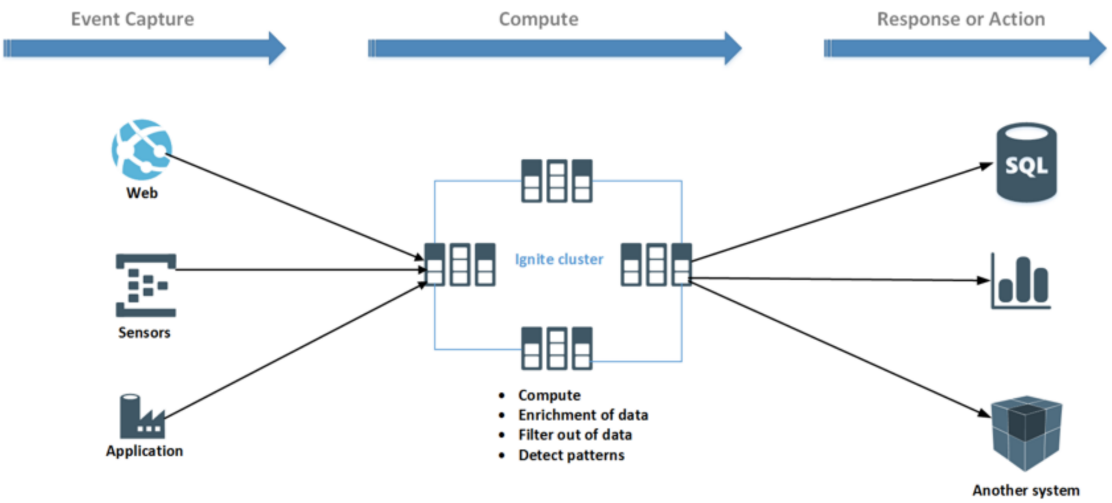

Figura 35: Fasi dello streaming ed elementi partecipanti

considerato il comportamento parallelo, tuttavia, è probabile che questi non giungano nello stesso ordine in cui sono stati inseriti nello stream.

Per ottenere una velocità ottimale, il data streamer utilizza le seguenti tecniche:

- le entry che devono arrivare allo stesso nodo vengono raggruppate in un buffer;
- più buffer possono coesistere nello stesso momento;
- per evitare di esaurire la memoria, un data streamer ha un numero massimo di buffer che può gestire contemporaneamente.

È anche possibile applicare una logica personalizzata e quindi processare i dati prima che questi vengano aggiunti alla cache, tramite uno *stream receiver*. Tale logica verrà applicata nel nodo in cui i dati dovranno essere memorizzati.

Per i dettagli implementativi su come utilizzare un data streamer e processare i dati tramite uno stream receiver fare riferimento al Capitolo 5.20.

3.6 Machine Learning

Con l'aggiunta del modulo di Machine Learning (ML) e di Deep Learning (DL), Ignite è in grado di offrire un insieme di strumenti che permettono di costruire modelli predittivi facilitando i data scientist sotto diversi aspetti. Il principale vantaggio si traduce nella possibilità di gestire tutto il flusso end-to-end in un'unico ambiente, eliminando la necessità di pesanti trasferimenti di dati che negli scenari tradizionali avvengono quando quest'ultimi devono essere spostati dalle sorgenti al luogo in cui avviene il processo di training vero e proprio, e quando alla fine bisogna fare il deploy nell'ambiente di produzione del modello ottenuto. In secondo luogo, la caratteristica propria di Ignite di poter

scalare orizzontalmente le risorse agevola gli algoritmi di ML e DL nella gestione dei dataset di grandi dimensioni, che possono talvolta superare la capacità di memorizzazione della singola macchina.

Un'altra conseguenza del gestire l'intero processo in un sistema unito dal punto di vista logico è quella di poter accedere sempre alla versione più recente dei dati: uno degli svantaggi delle tradizionali pipline di ML è infatti quello che il processo di training avviene molto spesso su dati ormai obsoleti. Questa caratteristica di Ignite, resa possibile grazie alla sua archiviazione incentrata sulla memoria, permette di eliminare le tempistiche legate alle operazioni di ETL necessarie quando si passa da un sistema all'altro. Come risultato, i framework di ML possono sfruttare a proprio vantaggio i dati che arrivano in real-time sul cluster ed implementare meccanismi di *continuous learning*.

Di seguito si elencano gli algoritmi disponibili suddivisi per task, assieme alle possibili applicazioni. Tali algoritmi sono ottimizzati per il processing distribuito e co-locato di Ignite.

- **Classificazione**
  - Casi d'uso: spam detection, image recognition, credit scoring, disease identification.
  - Algoritmi: Logistic Regression, Linear SVM (Support Vector Machine), k-NN Classification, Naive Bayes, Decision Trees, Random Forest, Multilayer Perceptron, Gradient Boosting, ANN (Approximate Nearest Neighbor).
- **Regressione**
  - Casi d'uso: stock prices, supermarket revenue.
  - Algoritmi: Linear Regression, Decision Trees Regression, k-NN Regression.
- **Clustering**
  - Casi d'uso: customer segmentation, shopping item clustering.
  - Algoritmi: K-Means Clustering, Gaussian mixture.
- **Recommendation**
  - Casi d'uso: playlist generation, product recommendation.
  - Algoritmi: Matrix Factorization.
- **Preprocessing**
  - Casi d'uso: data transformation, feature extraction, data normalization.
  - Algoritmi: Normalization Preprocessor, One-Hot-Encoder, Min-Max Scaler, pre-processing personalizzati tramite Partition-Based Dataset.

Il *Partition-Based Dataset* è un livello di astrazione sopra quello di memorizzazione e computazione di Ignite che permette di creare algoritmi che seguono i principi di zero ETL e fault tolerance. La filosofia alla base è aderente all'approccio MapReduce: ogni dataset viene diviso in partizioni, ciascuna delle quali è il punto di partenza di

un'operazione di map, i cui risultati confluiscono nell'operazione finale di reduce. Nel caso in cui un nodo venga a mancare durante la fase di learning, tutte le procedure di recovery sono trasparenti all'utente e il processo non si interrompe.

3.7 Messaging

Apache Ignite mette a disposizione un servizio di messaging *topic-based* e distribuito sui nodi del cluster. Tale servizio segue il paradigma *publish-subscribe*, che prevede la presenza di due entità: i publisher che inviano messaggi e i subscriber che li ricevono. I topic possono essere visti come delle etichette applicate ai messaggi, che ne descrivono la classe di appartenenza. I subscriber si iscrivono ad uno o più topic e ricevono solo i messaggi aventi come etichetta i topic a cui si sono iscritti. Il publisher è colui che pubblica i messaggi e gli assegna uno specifico topic. In questa maniera, i messaggi di una data tipologia verranno consegnati solo al sottoinsieme di subscriber che hanno effettivo interesse nel riceverli, dal momento che si sono registrati al rispettivo topic.

Il principale vantaggio di questo paradigma è che publisher e subscriber sono tra di loro debolmente accopiati, non avendo bisogno di conoscere l'esistenza l'uno dell'altro. Da un lato, i publisher non indirizzano i messaggi esplicitamente verso i subscriber, ma si limitano ad assegnare loro il topic di appartenenza. Dall'altro lato, i subscriber non necessitano di sapere quali siano i publisher in azione, perché si mettono semplicemente in ascolto dei messaggi appartenenti alle classi di loro interesse.

Questo sistema di messaggistica viene reso da Ignite distribuito: quando un nodo invia un messaggio m per il topic T, il messaggio viene pubblicato su tutti i nodi che si sono registrati a T. Ogni nuovo nodo che si unisce al cluster viene automaticamente iscritto a tutti i topic a cui gli altri nodi nel cluster, o nel cluster group a cui è stato assegnato il nuovo nodo, si sono registrati.

Per i dettagli implementativi fare riferimento al Capitolo 5.21.

3.8 Apache Hadoop performance acceleration

Per una trattazione più approfondita di Apache Hadoop consultare il Capitolo 7.1.1.

Ignite permette di migliorare le performance di Hadoop grazie al suo storage in-memory, che riduce di molto i tempi di accesso ai dati eliminando gli spostamenti dal disco o attraverso rete. Come risultato, i job possono essere avviati in una manciata di millisecondi piuttosto che in una decina di secondi, senza dover peraltro apportare modifche al codice.

L'acceleratore in-memory di Hadoop fornito da Ignite è utile nei casi in cui si ha già a disposizione un cluster Hadoop attivo e in esecuzione e si vogliono ottenere performance migliori col minimo sforzo. La sua implementazione si basa su tre componenti principali:

- **In-memory MapReduce**: si tratta di un'implementazione in-memory dell'engine MapReduce, completamente compatibile con HDFS e Yarn di Hadoop. È in grado di ridurre il tempo di avvio e di esecuzione dei tracker dei job e dei task, offrendo un'elaborazione distribuita a bassa latenza adatta per i task CPU-intensive.

Questo modulo fornisce anche una gestione di MapReduce basata sui pesi, i quali indicano quante risorse sono necessarie per eseguire un task e quindi quanti mapper e reducer utilizzare.

- **Ignite in-memory file system** (**IGFS**): costituisce un'alternativa al file system di Hadoop (HDFS) che permette di memorizzare i dati in memoria, minimizzando le operazioni di I/O col disco e migliorando quindi le performance. È in grado di fornire scalabilità e supporto alla gestione degli errori. Una delle limitazioni più grandi è rappresentata dalla necessità di spostare manualmente tutti i dati esistenti da HDFS a IGFS.

- **Hadoop file system cache**: allo scopo di risolvere la limitazione sopra citata, che costringe a spostare tutti i dati da un file system all'altro, questo modulo agisce come una cache in-memory distribuita di secondo livello al di sopra di HDFS, attraverso la quale passa ogni operazione di lettura e scrittura, migliorandone le performance. In questa maniera, IGFS può memorizzare in memoria solamente i blocchi più frequentemente usati dei dati, lasciando i file interi, che possono essere molto grandi, su HDFS.

3.9 Apache Spark performance acceleration

Per una trattazione più approfondita di Apache Spark consultare il Capitolo 7.1.2.

Ignite fornisce una sua implementazione degli RDD (**IgniteRDD**) che permette ad ogni dato o stato di essere condiviso in memoria tra i diversi job, worker, o applicazioni tramite un RDD condiviso. Gli RDD nativi di Spark, infatti, non consentono questo tipo di condivisione, dal momento che tali strutture sono locali a ciascun job. Dal punto di vista implementativo, un RDD condiviso di Ignite è implementato come se fosse una vista su una cache: tutte le modifche a quest'ultima sono subito visibili anche nello RDD. Gli RDD di Spark costituiscono invece delle collezioni immutabili di oggetti, pertanto qualsiasi modifica che li riguardi deve per forza passare attraverso la creazione di un nuovo RDD.

In maniera simile, anche relativamente all'altra struttura con cui lavora Spark, il DataFrame, Ignite offre dei miglioramenti in termini di tempistiche di accesso e di condivisione di dati tra job.

Infine, nonostante Spark fornisca una ricca sintassi SQL, non supporta l'indicizzazione. Ignite permette di impostare gli indici aumentando esponenzialmente le performance.

4 Algoritmi distribuiti utilizzati

4.1 Rendezvous Hashing

L'algoritmo di **Rendezvous Hashing**, noto anche come **Highest Random Weight (HRW) Hashing**, è un algoritmo distribuito di consenso che viene impiegato tipicamente per permettere ai client di raggiungere un accordo circa la posizione di un dato oggetto in un insieme di nodi. Detto in altri termini, consente di risolvere il problema dell'implementazione di una hash table in ambiente distribuito. Apache Ignite sfrutta tale algoritmo per capire come mappare le partizioni (che rappresentano un insieme di data entry) sui diversi nodi del cluster.

Prima di procedere con la descrizione dettagliata del funzionamento dell'algoritmo in oggetto, è utile capire perché altri algoritmi, a partire da quelli più semplici, non sono adatti a una situazione distribuita e quali sono i relativi svantaggi e criticità.

Approccio naïf La soluzione più semplice al problema consiste nell'applicare una hash function $h()$ all'oggetto che rappresenta la chiave in modo che venga identificato attraverso un valore numerico che viene poi diviso per N, ovvero per il numero di nodi del cluster. Il resto (modulo) della divisione determina il nodo in cui quella chiave verrà posizionata. Questo algoritmo ha il vantaggio di essere molto semplice e di garantire una distribuzione uniforme tra i vari nodi, tuttavia mostra le sue criticità nel caso in cui un nuovo nodo si aggiunga alla topologia. In tale situazione, infatti, cambiando il numero di nodi disponibili, cambierà la distribuzione delle chiavi già precedentemente assegnate, non solo verso il nuovo nodo, ma anche tra i nodi già precedentemente presenti nel cluster. Perciò, il requisito fondamentale di minima interruzione non viene rispettato. Questo comportamento è ben rappresentato dall'esempio in Figura 36, che mostra la distribuzione di 16 chiavi tra 3 nodi e la successiva ridistribuzione a seguito dell'aggiunta di un quarto nodo.

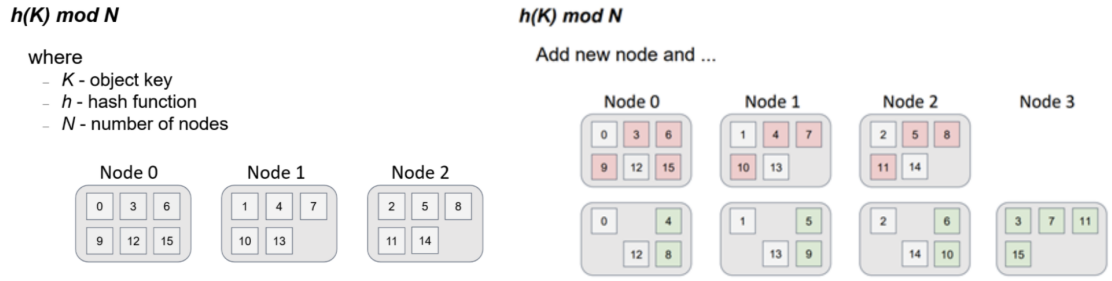

Figura 36: A sinistra la ripartizione originale delle chiavi nei nodi, a destra la successiva ridistribuzione a seguito dell'aggiunta di un nodo, con le chiavi soggette a uno spostamento colorate in verde

Consistent Hashing L'idea alla base di questo algoritmo è quella di mappare i nodi e gli oggetti da memorizzare in uno spazio di identificazione condiviso. Per farlo, si applica

alle chiavi degli oggetti la stessa hash function usata per gli identificatori dei nodi. Rappresentando lo spazio di identificazione come un cerchio, per identificare il nodo in cui un certo oggetto si trova, basta prendere il valore della hash function dalla chiave dell'oggetto e percorrere il cerchio in senso orario finché non si trova la corrispondenza. A questo punto, se si aggiunge un nodo, uno dei settori precedentemente definiti verrà diviso in due parti e il nuovo nodo acquisirà le chiavi di una delle due parti. Tuttavia, considerato che questo approccio dipende fortemente dagli identificativi dei nodi, gli oggetti non vengono distribuiti in maniera uniforme tra i nodi, come si può vedere dalla Figura 37.

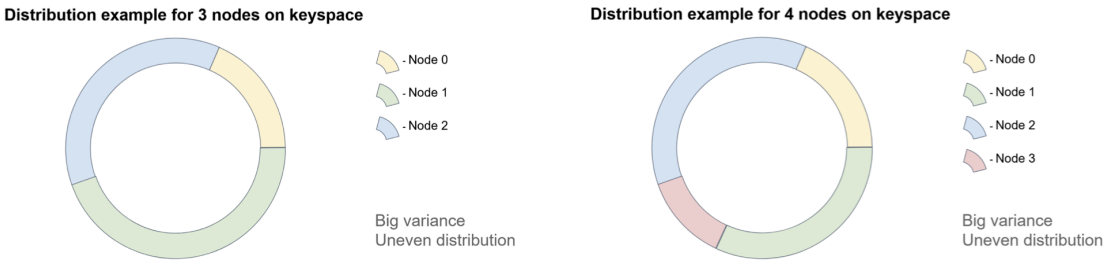

Figura 37: Spazio identificativo condiviso tra i nodi e le chiavi degli oggetti da memorizzare

Per far fronte a questo problema, è quindi possibile assegnare non uno ma molteplici valori di hash function per ciascun nodo, tipicamente sull'ordine delle centinaia, andando ad introdurre un insieme di hash function per ciascun nodo oppure applicando ricorsivamente la stessa hash function per ogni nodo, collezionando il risultato di ogni livello innestato. In questo caso, è necessario fare attenzione anche a possibili collisioni, dal momento che due nodi diversi non possono avere lo stesso identificativo.

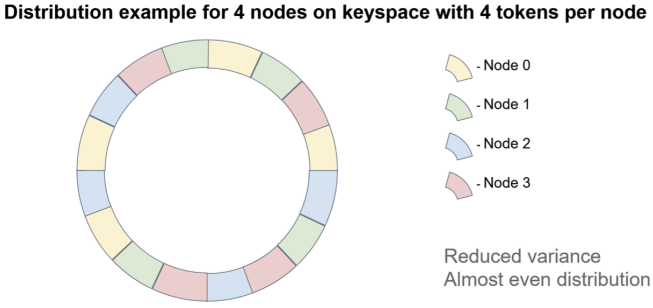

Figura 38: Spazio identificativo condiviso tra i nodi e le chiavi degli oggetti da memorizzare usando 4 funzioni hash per ciascun nodo

Nell'esempio in Figura 38, ciascun nodo è presente 4 volte nello spazio identificativo, con gli $N - 1$ nodi successivi al primo che fungono da repliche. In questo caso, il

requisito di minima interruzione viene rispettato poiché l'ordinamento nel cerchio rimane invariato, anche quando un nodo viene eliminato.

Rendezvous Hashing Questo algoritmo rappresenta in realtà una versione semplificata del precedente. Anche in questo caso, il principio valido alla base è quello di stabilità di ordinamento, ma invece di rendere i nodi e gli oggetti direttamente comparabili tra di loro, viene garantita la comparabilità solo a livello di uno specifico oggetto (chiave), indipendentemente dagli altri. Si definisce, quindi, il peso di un nodo in relazione a un certo oggetto, tramite una hash function, e si itera questo processo per tutti i nodi, ottenendo un insieme di pesi rispetto cui ordinare i nodi per la singola chiave. Il nodo con il peso maggiore sarà responsabile della memorizzazione della specifica chiave (Figura 39).

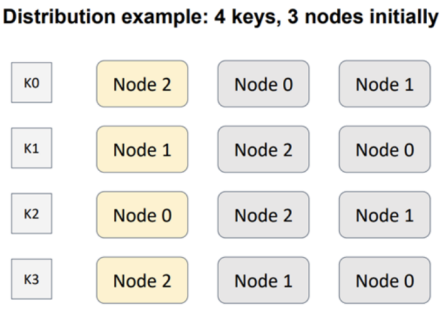

Figura 39: Distribuzione di 4 chiavi su 3 nodi: in giallo i nodi con il peso maggiore e quindi responsabili della specifica chiave

Se si suppone di aggiungere un quarto nodo alla topologia, la lista dei nodi per ciascuna chiave cambia. Facendo riferimento alla Figura 40 a sinistra, la posizione dei nodi colorati di rosso è cambiata rispetto a prima, in quanto il loro peso è inferiore rispetto a quello del nuovo nodo, colorato di verde. Solamente la chiave $K3$ viene quindi influenzata da tale modifica.

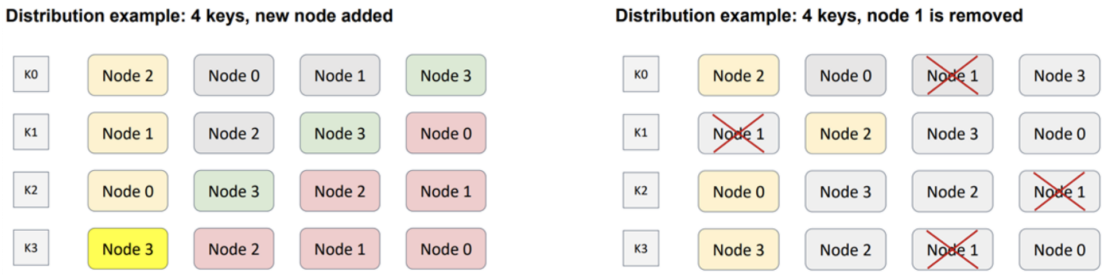

Figura 40: Aggiunta di un nodo (a sinistra) e rimozione di un nodo (a destra): in entrambi i casi tale cambiamento di topologia interessa una sola chiave

Allo stesso modo, se viene rimosso il quarto nodo (Figura 40 a destra) solo una chiave ne risente, questa volta $K1$. Le altre rimangono dove sono, dal momento che le rela-

zioni di ordinamento in ogni coppia di nodi rimangono invariate, come accadeva per il Consistent Hashing. Per questo motivo, il requisito di minima interruzione viene ancora una volta rispettato, ma a differenza del precedente algoritmo non vengono richiesti step aggiuntivi, come l'applicazione di hash function multiple per ciascun nodo. Se nel cluster è attiva la replicazione, ogni nodo successivo nella lista rappresenta il prossimo candidato ad ospitare una replica per quel determinato oggetto.

4.2 Two-Phase Commit Protocol (2PC)

L'algoritmo **Two-Phase Commit Protocol** (**2PC**) è un protocollo distribuito di consenso in grado di coordinare tutti i processi coinvolti in una transazione atomica distribuita e decidere se procedere con il commit (in caso di successo) o con il rollback (in caso di fallimento) della transazione. Nel contesto in cui opera Ignite, infatti, una singola transazione si estende su diversi nodi server e per garantire la data consistency, anche a livello di copie di backup, è necessario che nessuna parte coinvolta subisca un'interruzione o una perdita di dati.

Il protocollo provvede a designare un nodo coordinatore che darà il via al processo comunicando con i restanti nodi che assumono il ruolo di partecipanti. In caso si verifichino delle failure e tale processo non vada a buon fine, viene sfruttato un opportuno meccanismo di logging per gestire la procedura di recovery e far ritornare il sistema nello stato consistente in cui si trovava prima dell'inizio della transazione distribuita. A tale scopo, ogni nodo mantiene nel proprio data storage un write-ahead log, di modo che i dati qui contenuti non vengano mai persi o corrotti da un crash.

Come il nome stesso suggerisce, il protocollo gestisce ogni transazione in due fasi:

1. **Commit-request phase** (o **voting phase**): ogni volta che un client fa una richiesta di commit a un nodo, il nodo coordinatore invia un messaggio ai primary node per i dati di interesse. Ciascun nodo primario, dopo aver acquisito le lock necessarie, invia a sua volta un messaggio sincrono a ciascun nodo che mantiene una copia di backup dei dati. Ricevuto il messaggio, i nodi partecipanti eseguono la porzione di transazione che compete loro, memorizzandola nel log. A questo punto, ogni partecipante risponde con un messaggio di agreement (*yes*) nel caso la loro azione sia riuscita, altrimenti con un messaggio di abort (*no*) se si è verificata una failure che non permetta di procedere con il commit.

2. **Commit phase** (o **completion phase**): se durante la fase precedente il coordinatore ha ricevuto un messaggio di agreement da tutti i partecipanti procede ad inviare loro un messaggio di commit, ricevuto il quale ogni partecipante completa la propria azione, rilasciando tutte le lock e le risorse acquisite durante la transazione. Dopo che il coordinatore ha ricevuto un messaggio di acknowledgement da ogni partecipante, completa la transazione con successo. Nel caso in cui qualche partecipante abbia "votato no" durante la fase precedente, o non abbia inviato alcun messaggio, il coordinatore invia un messaggio di rollback a tutti i partecipanti, i quali annullano la propria porzione di transazione usando il log, rilasciano tutte le lock e le risorse precedentemente acquisite e restituiscono un messaggio

di acknowledgement al coordinatore, che procede ad annullare la transazione una volta ricevuti tutti i messaggi di risposta.

L'algoritmo appena descritto non è esente da difetti. Il principale svantaggio sta nel fatto di essere un protocollo bloccante: ciascun partecipante deve infatti aspettare la risposta del coordinatore (che sia di commit o di rollback) per poter procedere e nel caso il coordinatore fallisca in modo permanente, non saranno in grado di risolvere le proprie transazioni. In Figura 41 vengono schematizzate le azioni del protocollo in caso di successo.

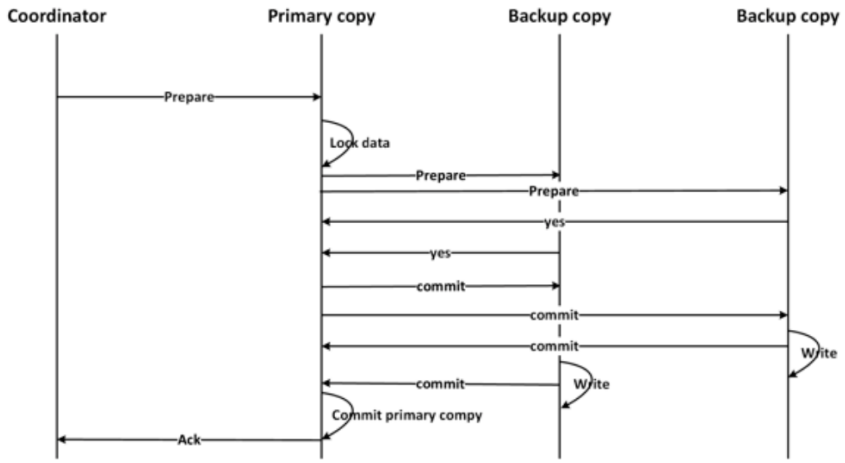

Figura 41: Comportamento del 2PC protocol nel caso la transazione distribuita avvenga successo

5 Configurazioni

In questo capitolo vengono riportati degli esempi di implementazione relativi ai concetti e alle funzionalità discusse nelle sezioni precedenti.

Per quanto riguarda gli aspetti di configurazione del cluster è possibile scegliere se specificarli attraverso un apposito file di configurazione in formato Spring XML[6] oppure programmaticamente, ad esempio tramite codice Java. Nel primo caso, al momento dell'avvio di un nodo dovrà essere passato come parametro il corrispondente file di configurazione personalizzato. Nel seguito, in alcuni casi vengono mostrate entrambe le versioni per poterle confrontare.

5.1 Cluster group

Per i relativi dettagli teorici fare riferimento al Capitolo 2.1.3.

Creazione di un cluster group formato da tutti i nodi remoti presenti nel cluster (escluso quindi il nodo locale) e successivo invio in broadcast di un job da eseguire presso di essi. Si fa notare che già l'interfaccia `IgniteCluster` rappresenta di per sé un gruppo di cluster, contenente la totalità dei nodi (incluso quello locale).

```
Ignite ignite = Ignition.ignite(); // get an instance of default no-name grid
IgniteCluster cluster = ignite.cluster();
// Get compute instance which will only execute
// over remote nodes, i.e. all the nodes except for this one
IgniteCompute compute = ignite.compute(cluster.forRemotes());
// Broadcast to all remote nodes and print the ID of the node
// on which this closure is executing
compute.broadcast(
        () -> System.out.println("Hello Node: " +
            ignite.cluster().localNode().id()));
```

Ignite definisce diversi cluster group predefiniti. Tra i più utilizzati si possono annoverare:

```
IgniteCluster cluster = ignite.cluster();
// All nodes on which the cache with name "myCache" is deployed,
// either in client or server mode
ClusterGroup cacheGroup = cluster.forCacheNodes("myCache");
// All data nodes responsible for caching data for "myCache"
ClusterGroup dataGroup = cluster.forDataNodes("myCache");
// All client nodes that can access "myCache"
ClusterGroup clientGroup = cluster.forClientNodes("myCache");
```

5.2 Meccanismi di discovery e configurazione di rete

Per i relativi dettagli teorici fare riferimento al Capitolo 2.1.3.

Come implementazione di default del meccanismo di TCP/IP Discovery, Ignite utilizza la classe `TcpDiscoverySpi`. Per quanto riguarda la modalità specifica con cui vengono intercettati gli indirizzi IP degli altri nodi, è possibile scegliere tra `TcpDiscoveryMulticastIpFinder`,

che come suggerisce il nome adotta un approccio multicast, e `TcpDiscoveryVmIpFinder`, che implementa una ricerca statica degli IP sulla base dell'insieme di indirizzi e di porte specificate. Sebbene sia consigliabile specificare almeno due o tre indirizzi, l'indicazione di anche un solo IP è sufficiente per permettere al cluster di formarsi correttamente: una volta che viene stabilita la connessione con tale indirizzo, infatti, Ignite è in grado di scoprire automaticamente tutti gli altri nodi.

Configurazione multicast via XML:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
  <property name="discoverySpi">
    <bean class="org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi">
      <property name="ipFinder">
        <bean class="org.apache.ignite.spi.discovery.tcp.ipfinder.multicast.TcpDiscoveryMulticastIpFinder">
          <property name="multicastGroup" value="228.10.10.157"/>
        </bean>
      </property>
    </bean>
  </property>
</bean>
```

Configurazione multicast in maniera programmatica:

```
TcpDiscoverySpi spi = new TcpDiscoverySpi();
TcpDiscoveryMulticastIpFinder ipFinder = new TcpDiscoveryMulticastIpFinder();
ipFinder.setMulticastGroup("228.10.10.157");
spi.setIpFinder(ipFinder);
IgniteConfiguration cfg = new IgniteConfiguration();
cfg.setDiscoverySpi(spi); // override default discovery SPI
Ignite ignite = Ignition.start(cfg); // start the node
```

Configurazione statica via XML:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
  <property name="discoverySpi">
    <bean class="org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi">
      <property name="ipFinder">
        <bean class="org.apache.ignite.spi.discovery.tcp.ipfinder.vm.TcpDiscoveryVmIpFinder">
          <!-- List of static IP addresses-->
          <property name="addresses">
            <list>
              <!-- Explicitly specifying address of a local node to let it start and
              operate normally even if there is no more nodes in the cluster.
              You can also optionally specify an individual port or port range. -->
              <value>1.2.3.4</value>
              <!-- IP address and optional port range of a remote node.
              You can also optionally specify an individual port. -->
              <value>1.2.3.5:47500..47509</value>
            </list>
          </property>
        </bean>
      </property>
    </bean>
  </property>
</bean>
```

È anche possibile usare congiuntamente il discovery multicast e statico. In questo caso, oltre agli indirizzi ricevuti in multicast, `TcpDiscoveryMulticastIpFinder` può anche lavorare con una lista pre-configurata di indirizzi IP statici, nel modo seguente:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
  <property name="discoverySpi">
    <bean class="org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi">
      <property name="ipFinder">
        <bean class="org.apache.ignite.spi.discovery.tcp.ipfinder.multicast.TcpDiscoveryMulticastIpFinder">
          <property name="multicastGroup" value="228.10.10.157"/>
          <property name="addresses">
            <list>
              <value>1.2.3.4</value>
              <value>1.2.3.5:47500..47509</value>
            </list>
          </property>
        </bean>
      </property>
    </bean>
  </property>
</bean>
```

Nel caso si debba passare allo ZooKeeper Discovery, occorre configurare la classe `ZookeeperDiscoverySpi` in un modo simile a questo:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
  <property name="discoverySpi">
    <bean class="org.apache.ignite.spi.discovery.zk.ZookeeperDiscoverySpi">
      <property name="zkConnectionString"
          value="127.0.0.1:34076,127.0.0.1:43310,127.0.0.1:36745"/>
      <property name="sessionTimeout" value="30000"/>
      <property name="zkRootPath" value="/apacheIgnite"/>
      <property name="joinTimeout" value="10000"/>
    </bean>
  </property>
</bean>
```

I primi due parametri sono obbligatori e specificano rispettivamente la lista degli indirizzi dei nodi ZooKeeper e il lasso di tempo oltre il quale un nodo Ignite che non reagisce agli eventi di discovery viene considerato disconnesso.

Una volta che i nodi si sono reciprocamente trovati, e hanno quindi formato il cluster, iniziano a scambiarsi messaggi attraverso i quali vengono sottomesse varie operazioni distribuite come l'esecuzione di un task, la modifica dei dati o l'esecuzione di una query. L'implementazione che viene usata di default è quella di `TcpCommunicationSpi`, mediante la quale è possibile specificare varie proprietà (come l'indirizzo e la porta che ciascun nodo apre per la comunicazione e a cui gli altri nodi si connettono inviando messaggi, così come diversi timeout di connessione), in modo analogo ai precedenti esempi.

5.3 Data region

Per i relativi dettagli teorici fare riferimento al Capitolo 2.2.

Se si ha necessità di cambiare le proprietà della data region di default, o se si vogliono aggiungere altre regioni, è possibile farlo attraverso la `DataStorageConfiguration`. Nel seguente snippet viene mostrata la creazione di una data region personalizzata, che va ad aggiungersi a quella predefinita, per la quale si specifica la dimensione iniziale e quella massima consentita, oltre che la policy di eviction da utilizzare nel caso in cui si raggiunga il limite di memoria. Per mappare una certa cache alla regione appena definita è sufficiente esplicitarne il nome mediante la proprietà `dataRegionName` contestualmente alla configurazione della cache.

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration" id="ignite.cfg">
  <property name="dataStorageConfiguration">
    <bean class="org.apache.ignite.configuration.DataStorageConfiguration">
      <!-- Default memory region that grows endlessly. Any cache will be bound
      to this memory region unless another region is set in the cache's configuration. -->
      <property name="defaultDataRegionConfiguration">
        <bean class="org.apache.ignite.configuration.DataRegionConfiguration">
          <property name="name" value="Default_Region"/>
          <!-- 100 MB memory region with disabled eviction -->
          <property name="initialSize" value="#{100 * 1024 * 1024}"/>
        </bean>
      </property>
      <property name="dataRegionConfigurations">
        <list>
          <!-- 40MB memory region with eviction enabled -->
          <bean class="org.apache.ignite.configuration.DataRegionConfiguration">
            <property name="name" value="40MB_Region_Eviction"/>
            <!-- Memory region of 20 MB initial size -->
            <property name="initialSize" value="#{20 * 1024 * 1024}"/>
            <!-- Maximum size is 40 MB -->
            <property name="maxSize" value="#{40 * 1024 * 1024}"/>
            <!-- Enabling eviction for this memory region -->
            <property name="pageEvictionMode" value="RANDOM_2_LRU"/>
          </bean>
        </list>
      </property>
    </bean>
  </property>
  <property name="cacheConfiguration">
    <list>
      <!-- Cache that is mapped to a specific data region -->
      <bean class="org.apache.ignite.configuration.CacheConfiguration">
        <property name="name" value="SampleCache"/>
        <!-- Assigning the cache to the "40MB_Region_Eviction" region -->
        <property name="dataRegionName" value="40MB_Region_Eviction"/>
      </bean>
    </list>
  </property>
</bean>
```

5.4 Cache

Per i relativi dettagli teorici fare riferimento al Capitolo 2.2.1 e al Capitolo 2.4.5.

L'esempio seguente mostra come configurare una cache di nome `myCache` in modalità partizionata, con due copie di backup, in modalità di ribilanciamento sincrono, con gli aggiornamenti sui nodi primari e i nodi di backup che avvengono in maniera sincrona, e che è fruibile in modalità read-only.

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
   <property name="cacheConfiguration">
      <bean class="org.apache.ignite.configuration.CacheConfiguration">
         <property name="name" value="myCache"/>
         <property name="cacheMode" value="PARTITIONED"/>
         <property name="backups" value="2"/>
         <property name="rebalanceMode" value="SYNC"/>
         <property name="writeSynchronizationMode" value="FULL_SYNC"/>
         <property name="partitionLossPolicy" value="READ_ONLY_SAFE"/>
      </bean>
   </property>
</bean>
```

5.5 Cache group

Per i relativi dettagli teorici fare riferimento al Capitolo 2.2.1.

Per creare un cache group e associarlo a determinate cache è sufficiente impostare la proprietà `groupName`:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
   <property name="cacheConfiguration">
      <list>
         <bean class="org.apache.ignite.configuration.CacheConfiguration">
            <property name="name" value="Person"/>
            <property name="backups" value="1"/>
            <!-- Group the cache belongs to -->
            <property name="groupName" value="group1"/>
         </bean>
         <bean class="org.apache.ignite.configuration.CacheConfiguration">
            <property name="name" value="Organization"/>
            <property name="backups" value="1"/>
            <!-- Group the cache belongs to -->
            <property name="groupName" value="group1"/>
         </bean>
      </list>
   </property>
</bean>
```

La corrispondente operazione in modalità programmatica si presenta nel modo seguente:

```java
// Defining cluster configuration
IgniteConfiguration cfg = new IgniteConfiguration();

// Defining "Person" cache configuration
CacheConfiguration<Integer, Person> personCfg = new CacheConfiguration<Integer,
    Person>("Person");

personCfg.setBackups(1);

// Group the cache belongs to
personCfg.setGroupName("group1");

// Defining "Organization" cache configuration
CacheConfiguration orgCfg = new CacheConfiguration("Organization");

orgCfg.setBackups(1);

// Group the cache belongs to
orgCfg.setGroupName("group1");

cfg.setCacheConfiguration(personCfg, orgCfg);

// Starting the node
Ignition.start(cfg);
```

5.6 Native persistence

Per i relativi dettagli teorici fare riferimento al Capitolo 2.2.3.

Per abilitare la modalità di persistenza nativa, occorre impostare opportunamente la proprietà `persistenceEnabled` a livello di singola data region. Lo snippet seguente configura tale modalità per la data region di default.

```xml
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
   <property name="dataStorageConfiguration">
      <bean class="org.apache.ignite.configuration.DataStorageConfiguration">
         <property name="defaultDataRegionConfiguration">
            <bean class="org.apache.ignite.configuration.DataRegionConfiguration">
               <property name="persistenceEnabled" value="true"/>
            </bean>
         </property>
      </bean>
   </property>
</bean>
```

5.7 Key-Value API

Per i relativi dettagli teorici fare riferimento al Capitolo 2.4.1.

Di seguito vengono mostrate alcune delle principali operazioni atomiche che possono essere eseguite su una cache.

```java
Ignite ignite = Ignition.ignite();
IgniteCache<Long, String> cache = ignite.cache("myCache");

// Stores keys in the cache (the values will end up on different cache nodes)
for (int i = 0; i < 10; i++)
    cache.put(i, Integer.toString(i));
// Gets keys stored in the cache
for (int i = 0; i < 10; i++)
    System.out.println("Got [key=" + i + ", val=" + cache.get(i) + ']');
// Stores given key-value pair in cache only if cache had no previous mapping for
    it and returns boolean success flag
boolean success = cache.putIfAbsent(11, "Hello");
// As above, but returns previous value associated with the key, if present
String oldVal = cache.getAndPutIfAbsent(22, "World");
// Opposite of getAndPutIfAbsent(), returns previous value
oldVal = cache.getAndReplace(11, "New value");
// Opposite of putIfAbsent(), returns boolean success flag
success = cache.replace(22, "Other new value");
// Replace with the last specified value if the given key-value pair (first two
    arguments) exists, returns boolean success flag
success = cache.replace(22, "Other new value", "Yet-another-new-value");
// Removes key-value pair if exists, returns boolean success flag
success = cache.remove(11, "Hello");
```

Esistono anche metodi come `getAll()`, `putAll()` e `removeAll()` che consentono di eseguire operazioni bulk. Tuttavia, visto che sono in realtà eseguite come sequenza di operazioni atomiche, tali metodi possono fallire parzialmente generando una `CachePartialUpdateException` contenente la lista delle chiavi per cui l'aggiornamento è fallito.

5.8 Affinity colocation

Per i relativi dettagli teorici fare riferimento al Capitolo 2.4.4.

Prendendo come riferimento uno scenario nel quale si hanno degli ordini (tabella `Order`) composti da un certo numero di articoli (tabella `OrderItem`), un modo per co-locare gli articoli sullo stesso nodo in cui si trova l'ordine corrispondente consiste nel definire l'oggetto complesso `OrderItemKey` che rappresenta l'affinity key composta dall'identificatore dell'articolo e dal campo che si vuole usare per la co-locazione (ovvero l'identificatore dell'ordine), opportunamente annotato con `@AffinityKeyMapped`.

```java
public class AffinityColocationExample {
    static class OrderItem {
        private int id;
        private String orderId;
        private String name;

        public OrderItem(int id, int orderId, String name) {
            this.id = id;
            this.orderId = orderId;
            this.name = name;
        }

        public int getId() {
            return id;
        }
    }

    static class OrderItemKey {
        private int id;
        @AffinityKeyMapped
        private int orderId;

        public OrderItemKey(int id, int orderId) {
            this.id = id;
            this.orderId = orderId;
        }
    }

    static class Order {
        private int id;
        private String name;

        public Order(int id, String name) {
            this.id = id;
            this.name = name;
        }

        public int getId() {
            return id;
        }
    }

    public void configureAffinityKeyWithAnnotation() {
        CacheConfiguration<OrderItemKey, OrderItem> orderItemCfg = new CacheConfiguration<>("order_item");
        orderItemCfg.setBackups(1);
        CacheConfiguration<Integer, Order> orderCfg = new CacheConfiguration<>("order");
        orderCfg.setBackups(1);

        try (Ignite ignite = Ignition.start()) {
            IgniteCache<OrderItemKey, OrderItem> orderItemCache = ignite.getOrCreateCache(orderItemCfg);
            IgniteCache<Integer, Order> orderCache = ignite.getOrCreateCache(orderCfg);

            Order o1 = new Order(1, "Order 1");
            OrderItem oi1 = new OrderItem(1, o1.getId(), "Item 1");

            // Both the o1 and oi1 objects will be cached on the same node
            orderItemCache.put(new OrderItemKey(oi1.getId(), o1.getId()), oi1);
            orderCache.put(1, o1);

            // Get the order item object
            oi1 = orderItemCache.get(new OrderItemKey(1, 1));
        }
    }
}
```

In alternativa alla definizione di `OrderItemKey`, è possibile sfruttare la classe `AffinityKey`, predisposta specificamente per definire un affinity mapping personalizzato.

5.9 Distributed computing

Per i relativi dettagli teorici fare riferimento al Capitolo 3.1.

Il punto di partenza per sfruttare le computazioni distribuite è l'interfaccia `IgniteCompute`, che si può ottenere a partire da un'istanza di `Ignite`. Questa interfaccia fornisce i metodi per la distribuzione dei task sui nodi del cluster e per sfruttare le *colocated computation* (vedi Capitolo 5.12). Ciascuna interfaccia viene associata a tutti i nodi o a un sotto-gruppo di essi (vedi Capitolo 5.1): i nodi specificati saranno quelli che eseguiranno i task. Un task può essere definito tramite una `IgniteRunnable`, una `IgniteCallable` o una `IgniteClosure`, delle quali sono disponibili anche le rispettive versioni asincrone. È possibile scegliere se eseguire il task una sola volta, su uno dei nodi del cluster, o se sottometterlo a tutti i nodi tramite il metodo `broadcast()`.

Esempio di task definito tramite interfaccia `IgniteRunnable`, che può essere usata per implementare computazioni che non richiedono parametri in ingresso e che non producono risultati:

```
IgniteCompute compute = ignite.compute();

// Iterate through all words and print
// each word on a different cluster node
for (String word : "Print words on different cluster nodes".split(" ")) {
    compute.run(() -> System.out.println(word));
}
```

Esempio di task definito tramite interfaccia `IgniteCallable`, che consente di restituire un valore in output:

```
Collection<IgniteCallable<Integer>> calls = new ArrayList<>();

// Iterate through all words in the sentence and create callable jobs
for (String word : "How many characters".split(" "))
    calls.add(word::length);

// Execute the collection of callables on the cluster
Collection<Integer> res = ignite.compute().call(calls);

// Add up all the word lengths received from cluster nodes
int total = res.stream().mapToInt(Integer::intValue).sum();
```

Esempio di task definito tramite interfaccia funzionale `IgniteClosure`, che accetta un parametro e restituisce un valore:

```
IgniteCompute compute = ignite.compute();

// Execute closure on all cluster nodes
Collection<Integer> res = compute.apply(String::length, Arrays.asList("How many
    characters".split(" ")));

// Add all the word lengths received from cluster nodes
int total = res.stream().mapToInt(Integer::intValue).sum();
```

Spesso è utile condividere lo stato di diversi compute job eseguiti sullo stesso nodo. Per fare ciò, viene resa disponibile una mappa locale concorrente e condivisa su ogni nodo. Concettualmente, questi dati locali sono simili alle variabili locali dei thread, dal momento che non vengono distribuiti ma rimangono all'interno dello scope locale del nodo.

```
IgniteCluster cluster = ignite.cluster();
ConcurrentMap<String, Integer> nodeLocalMap = cluster.nodeLocalMap();
```

Un altro modo di eseguire dei task è quello di appoggiarsi ad un `ExecutorSevice`. Ignite ne fornisce un'implementazione distribuita che si occupa di assegnare il carico di lavoro dei task in maniera bilanciata tra i nodi del cluster. Anche in questo caso, è possibile limitare l'insieme di nodi disponibili all'executor service specificando un cluster group.

```
// Get cluster-enabled executor service
ExecutorService exec = ignite.executorService();

// Iterate through all words in the sentence and create jobs
for (final String word : "Print words using runnable".split(" ")) {
    // Execute runnable on some node
    exec.submit(new IgniteRunnable() {
        @Override
        public void run() {
            System.out.println(">>> Printing '" + word + "' on this node from
                grid job.");
        }
    });
}
```

Affinché i task possano essere correttamente eseguiti sui nodi remoti, è necessario attivare la modalità di *peer class loading* in maniera programmatica:

```
IgniteConfiguration cfg = new IgniteConfiguration();
cfg.setPeerClassLoadingEnabled(true);
// Start the node
Ignite ignite = Ignition.start(cfg);
```

oppure tramite XML:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
   <property name="peerClassLoadingEnabled" value="true"/>
</bean>
```

5.10 MapReduce API

Per i relativi dettagli teorici fare riferimento al Capitolo 3.1.

Il paradigma MapReduce è applicabile mediante l'interfaccia `ComputeTask`, che fornisce i metodi di `map()`, di `result()` e di `reduce()`. Il primo metodo riceve in input l'insieme dei nodi del cluster su cui il task deve essere eseguito assieme ai parametri del task stesso e restituisce in output una mappa con i job da eseguire come chiave e i worker node che si fanno carico della sua computazione come valore. I job vengono quindi inviati ai nodi corrisponenti ed eseguiti. Il secondo metodo, invece, viene richiamato dopo che ciascun job viene completato e restituisce un'indicazione su come si deve procedere con il resto del task:

- `WAIT`: si aspetta che tutti gli altri job abbiano completato
- `REDUCE`: si passa immediatamente alla fase di reduce, scartando tutti i restanti job e i rispettivi risultati non ancora ricevuti
- `FAILOVER`: si passa il job ad un altro nodo per la riesecuzione, nel caso ad esempio abbia fallito

Il terzo metodo, infine, viene chiamato nella fase finale di reduce, quando tutti i job sono terminati o nel caso in cui la `result()` abbia restituito la policy di `REDUCE` per un certo job, ricevendo una lista con tutti i risultati parziali che verranno aggregati per ottenere il risultato finale.

Esistono tuttavia delle classi predefinite che forniscono le implementazioni più comunmente usate dei metodi `map()` e `result()`, lasciando allo sviluppatore il compito di definire unicamente la logica di reduce. Una di queste è la `ComputeTaskSplitAdapter` che implementa il metodo `result()` in modo che restituisca la policy di `FAILOVER` in caso di fallimento del job e quella `WAIT` in caso contrario. Introduce anche un nuovo metodo `split()` in grado di produrre i job sulla base dei dati in input. Di seguito viene mostrato un esempio che si serve di tale classe per calcolare il numero di caratteri di una stringa passando a ciascun nodo una differente parola. Il task in questione viene passato al metodo `execute()`, che prende come ultimo argomento il parametro di input del task stesso:

```java
public class ComputeTaskExample {
    public static class CharacterCountTask extends ComputeTaskSplitAdapter<String, Integer> {
        // 1. Splits the received string into words
        // 2. Creates a child job for each word
        // 3. Sends the jobs to other nodes for processing
        @Override
        public List<ComputeJob> split(int gridSize, String arg) {
            String[] words = arg.split(" ");
            List<ComputeJob> jobs = new ArrayList<>(words.length);
            for (final String word : words) {
                jobs.add(new ComputeJobAdapter() {
                    @Override
                    public Object execute() {
                        System.out.println(">>> Printing '" + word + "' on from compute
                            job.");

                        return word.length(); // return the number of letters in the word
                    }
                });
            }

            return jobs;
        }

        @Override
        public Integer reduce(List<ComputeJobResult> results) {
            int sum = 0;
            for (ComputeJobResult res : results)
                sum += res.<Integer>getData();

            return sum;
        }
    }

    public static void main(String[] args) {
        Ignite ignite = Ignition.start();
        IgniteCompute compute = ignite.compute();
        // Execute the task on the cluster and wait for its completion
        int cnt = compute.execute(CharacterCountTask.class, "Hello Grid Enabled World!");
        System.out.println(">>> Total number of characters in the phrase is '" + cnt + "'.");
    }
}
```

5.11 Task-data colocation

Per i relativi dettagli teorici fare riferimento al Capitolo 3.1.

Per eseguire un task presso i nodi in cui si trovano i dati di cui necessita si utilizzano i metodi `affinityRun()` o `affinityCall()` dell'interfaccia `IgniteCompute`. Tale co-locazione può avvenire sia per chiave che per partizione. Per determinare la posizione in cui si trova la data chiave o partizione, Ignite utilizza l'affinity function configurata.

Esempio di co-locazione per chiave, mediante il metodo `affinityRun()` a cui viene passato il task sotto forma di runnable:

```java
IgniteCache<Integer, String> cache = ignite.cache("myCache");
IgniteCompute compute = ignite.compute();
int key = 1;
// This closure will execute on the remote node where data for the given 'key' is located
compute.affinityRun("myCache", key, () -> {
    // Peek is a local memory lookup
    System.out.println("Co-located [key= " + key + ", value= " + cache.localPeek(key) + ']');
});
```

Usando il metodo `affinityCall()` sarà invece necessario modellare il task mediante una callable.

Nel caso in cui si debbano ottenere dei risultati da più chiavi che si sanno appartenere alla stessa partizione, conviene inviare il task direttamente al nodo che ospita quella specifica partizione, evitando di creare un task per ciascuna chiave. Per farlo, basta specificare l'identificatore della partizione al posto della chiave nei due metodi sopracitati.

5.12 Indici

Per i relativi dettagli teorici fare riferimento al Capitolo 3.2.

Il codice SQL per creare un indice è il seguente:

```
CREATE [SPATIAL] INDEX [[IF NOT EXISTS] indexName] ON tableName
   (columnName [ASC|DESC] [,...]) [(index_option [...])]

index_option := {INLINE_SIZE size | PARALLEL parallelism_level}
```

Tra le opzioni aggiuntive che si possono specificare contestualmente alla creazione, si trova `INLINE_SIZE` che permette di specificare la dimensione dell'indice in byte e `PARALLEL` che consente di indicare il numero di thread da usare in parallelo per la creazione dell'indice.

5.13 SQL API

Per i relativi dettagli teorici fare riferimento al Capitolo 3.2.

Per fare in modo che uno specifico campo sia effettivamente interrogabile con una query è necessario abilitarlo esplicitamente allo scopo, mediante l'annotazione `@QuerySqlField`. Attraverso di essa, è possibile configurare anche gli indici:

```java
public class Person implements Serializable {
    @QuerySqlField(index = true) // indexed field, will be visible to the SQL engine
    private long id;
    @QuerySqlField // queryable field, will be visible to the SQL engine
    private String name;
    private int age; // will NOT be visible to the SQL engine
    // Indexed field sorted in descending order, will be visible to the SQL engine
    @QuerySqlField(index = true, descending = true)
    private float salary;
}
```

In alternativa, è possibile usare la classe `QueryEntity`, utile soprattutto se si vuole specificare la configurazione tramite XML:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration" id="ignite.cfg">
   <property name="cacheConfiguration">
      <bean class="org.apache.ignite.configuration.CacheConfiguration">
         <property name="name" value="Person"/>
            <property name="queryEntities">
               <list>
                  <bean class="org.apache.ignite.cache.QueryEntity">
                     <property name="keyType" value="java.lang.Long"/>
                     <property name="keyFieldName" value="id"/>
                     <property name="valueType" value="org.apache.ignite.examples.Person"/>
                     <!-- Defining fields that will be either indexed or queryable.
                     Indexed fields are added to the 'indexes' list below. -->
                     <property name="fields">
                        <map>
                           <entry key="id" value="java.lang.Long"/>
                           <entry key="name" value="java.lang.String"/>
                           <entry key="salary" value="java.lang.Float "/>
                        </map>
                     </property>
                     <!-- Defining indexed fields -->
                     <property name="indexes">
                        <list>
                           <!-- Single field (aka column) index -->
                           <bean class="org.apache.ignite.cache.QueryIndex">
                              <constructor-arg value="name"/>
                           </bean>
                           <!-- Group index -->
                           <bean class="org.apache.ignite.cache.QueryIndex">
                              <constructor-arg>
                                 <list>
                                    <value>id</value>
                                    <value>salary</value>
                                 </list>
                              </constructor-arg>
                              <constructor-arg value="SORTED"/>
                           </bean>
                        </list>
                     </property>
                  </bean>
               </list>
            </property>
         </property>
      </bean>
   </property>
</bean>
```

Per l'esecuzione delle query viene fornita la classe `SqlFieldsQuery`, attraverso cui è possibile eseguire tutte le operazioni di DDL e di DML. Al costruttore viene passata la stringa della query in sintassi SQL, dopodiché questa viene eseguita mediante il metodo `query()` applicato alla cache che contiene i dati di riferimento e che restituisce un cursore attraverso cui iterare sui risultati dell'interrogazione. Per le operazioni di DDL è sufficiente richiamare il metodo `getAll()` sul cursore.

```
IgniteCache<Long, Person> cache = ignite.cache("personCache");

cache.query(
    new SqlFieldsQuery(
        "INSERT INTO Person(id, firstName, lastName) VALUES(?, ?, ?)")
        .setArgs(1L, "John", "Smith"))
        .getAll();

SqlFieldsQuery sql = new SqlFieldsQuery(
    "SELECT CONCAT(firstName, ' ', lastName) FROM Person");
// Iterate over the result set
try (QueryCursor<List<?>> cursor = cache.query(sql)) {
    for (List<?> row : cursor)
        System.out.println("personName=" + row.get(0));
}
```

È possibile forzare l'esecuzione in locale delle query, interrogando solo i dati contenuti in un singolo nodo, tramite l'impostazione `SqlFieldsQuery.setLocal(true)`. A meno di casi particolari, questa modalità produrrà sempre dei risultati incompleti.

5.14 Join distribuiti

Per i relativi dettagli teorici fare riferimento al Capitolo 3.2.

Per abilitare con successo la modalità di join non-colocato ed evitare di ottenere risultati non corretti, è necessario richiamare il metodo `setDistributedJoins(true)` sull'oggetto di tipo `SqlFieldsQuery` rappresentante la query di join.

5.15 Transazioni

Per i relativi dettagli teorici fare riferimento al Capitolo 3.2.

Le transazioni in SQL possono essere realizzate mediante i comandi di `BEGIN`, `COMMIT` e `ROLLBACK`.

Esempio di transazione che viene eseguita con successo e termina con il commit delle modifiche:

```
BEGIN;
INSERT INTO Person (id, name, city_id) VALUES (1, 'John Doe', 3);
UPDATE City SET population = population + 1 WHERE id = 3;
COMMIT;
```

Esempio di transazione che termina con un rollback e che pertanto non ha effetto:

```
BEGIN;
INSERT INTO Person (id, name, city_id) VALUES (1, 'John Doe', 3);
UPDATE City SET population = population + 1 WHERE id = 3;
ROLLBACK;
```

5.16 Comandi operazionali

Ignite mette a disposizione una serie di comandi operazionali che permettono di interagire con file, esecuzioni di query, task e job. I comandi disponibili sono i seguenti:

- **COPY**: permette di copiare dati da un file CSV a una tabella SQL.

```
COPY FROM '/path/to/local/file.csv'
INTO tableName (columnName1, columnName2, ...) FORMAT CSV [CHARSET '<charset-name>']
```

- **SET STREAMING**: effettua uno streaming dei dati in modalità bulk da un file CSV a una tabella SQL. Per farlo, basta preparare un file con il comando `SET STREAMING ON` seguito dai comandi `INSERT` per i dati che devono essere caricati, le cui tabelle devono essere già presenti nel cluster. Altri comandi diversi dall'inserimento non sono ammessi.

- **KILL QUERY**: permette di cancellare una query in esecuzione. Questo comando va ad agire su tutti i nodi in cui viene eseguita.

```
KILL QUERY [ASYNC] 'query_id'
```

- **KILL TRANSACTION**: permette di cancellare una transazione in corso.

```
KILL TRANSACTION 'xid'
```

- **KILL SCAN**: permette di cancellare una scan query in corso.

```
KILL SCAN 'origin_node_id' 'cache_name' query_id
```

- **KILL COMPUTE**: permette di cancellare una computazione in corso.

```
KILL COMPUTE 'session_id'
```

- **KILL CONTINUOUS**: permette di cancellare una continous query in esecuzione.

```
KILL CONTINUOUS 'origin_node_id', 'routine_id'
```

- **KILL SERVICE**: permette di cancellare un servizio in esecuzione.

```
KILL SERVICE 'name'
```

5.17 Continuos query

Per i relativi dettagli teorici fare riferimento al Capitolo 3.3.

Aggiunta di un local listener ad una continuous query:

```
IgniteCache<Integer, String> cache = ignite.getOrCreateCache("myCache");

ContinuousQuery<Integer, String> query = new ContinuousQuery<>();
query.setLocalListener(new CacheEntryUpdatedListener<Integer, String>() {
    @Override
    public void onUpdated(Iterable<CacheEntryEvent<? extends Integer, ? extends String>>
        events)
        throws CacheEntryListenerException {
        // react to the update events here
    }
});

cache.query(query);
```

Registrazione di una initial query che verrà eseguita prima della continuous query:

```
IgniteCache<Integer, String> cache = ignite.getOrCreateCache("myCache");

ContinuousQuery<Integer, String> query = new ContinuousQuery<>();
// Setting an optional initial query.
// The query will return entries for the keys greater than 10.
query.setInitialQuery(new ScanQuery<>((k, v) -> k > 10));
// Mandatory local listener
query.setLocalListener(events -> {});

try (QueryCursor<Cache.Entry<Integer, String>> cursor = cache.query(query)) {
    // Iterating over the entries returned by the initial query
    for (Cache.Entry<Integer, String> e : cursor)
        System.out.println("key=" + e.getKey() + ", val=" + e.getValue());
}
```

Impostazione di un remote filter sulla continuous query:

```
ContinuousQuery<Integer, String> qry = new ContinuousQuery<>();

qry.setLocalListener(events ->
    events.forEach(event -> System.out.format("Entry: key=[%s] value=[%s]\n",
        event.getKey(), event.getValue()))
);

qry.setRemoteFilterFactory(new Factory<CacheEntryEventFilter<Integer, String>>() {
    @Override
    public CacheEntryEventFilter<Integer, String> create() {
        return new CacheEntryEventFilter<Integer, String>() {
            @Override
            public boolean evaluate(CacheEntryEvent<? extends Integer, ? extends String> e) {
                System.out.format("the value for key [%s] was updated from [%s] to [%s]\n",
                    e.getKey(), e.getOldValue(), e.getValue());
                return true;
            }
        };
    }
});
```

Impostazione di un remote transformer:

```
IgniteCache<Integer, Person> cache = ignite.getOrCreateCache("myCache");
// Create a new continuous query with a transformer
ContinuousQueryWithTransformer<Integer, Person, String> qry = new
    ContinuousQueryWithTransformer<>();
// Factory to create transformers
Factory factory = FactoryBuilder.factoryOf(
    // Return one field of a complex object.
    // Only this field will be sent over to the local listener.
    (IgniteClosure<CacheEntryEvent, String>)
        event -> ((Person)event.getValue()).getName()
);

qry.setRemoteTransformerFactory(factory);

// Listener that will receive transformed data
qry.setLocalListener(names -> {
    for (String name : names)
        System.out.println("New person name: " + name);
});
```

5.18 Transazioni distribuite

Per i relativi dettagli teorici fare riferimento al Capitolo 3.4.

Di seguito si riporta un esempio che mostra come aggiungere il supporto transazionale alle cache e come utilizzare l'API per eseguire una transazione:

```
Ignite ignite = Ignition.start()

CacheConfiguration<Integer, Person> personCfg = new CacheConfiguration<>("Person");
personCfg.setAtomicityMode(CacheAtomicityMode.TRANSACTIONAL);
IgniteCache<Integer, Person> personCache = ignite.getOrCreateCache(personCfg)

CacheConfiguration<Integer, City> cityCfg = new CacheConfiguration<>("City");
cityCfg.setAtomicityMode(CacheAtomicityMode.TRANSACTIONAL);
IgniteCache<Integer, City> cityCache = ignite.getOrCreateCache(cityCfg)

IgniteTransactions transactions = ignite.transactions();
try (Transaction tx = transactions.txStart()) {
    personCache.put(1, new Person(1, "John", "Doe", 304));

    City c = cityCache.get(304);
    cityCache.put(304, new City(c.getId(), c.getName(), c.getPopulationCount() + 1);

    tx.commit();
}
```

5.19 Modalità concorrente e livelli di isolamento

Per i relativi dettagli teorici fare riferimento al Capitolo 3.4.

Esempio di impostazione della concurrency mode e dell'isolation level:

```java
CacheConfiguration<Integer, String> cfg = new CacheConfiguration<>();
cfg.setAtomicityMode(CacheAtomicityMode.TRANSACTIONAL);
cfg.setName("myCache");
IgniteCache<Integer, String> cache = ignite.getOrCreateCache(cfg);
// Re-try the transaction a limited number of times
int retryCount = 10;
int retries = 0;
// Start a transaction in the OPTIMISTIC mode with the SERIALIZABLE isolation level
while (retries < retryCount) {
    retries++;
    try (Transaction tx = ignite.transactions().txStart(TransactionConcurrency.OPTIMISTIC,
            TransactionIsolation.SERIALIZABLE)) {
        // Modify cache entries as part of this transaction
        cache.put(1, "foo");
        cache.put(2, "bar");
        tx.commit(); // commit the transaction
        // The transaction succeeded. Leave the while loop.
        break;
    } catch (TransactionOptimisticException e) {
        // Transaction has failed. Retry.
    }
}
```

5.20 Real time streaming API

Per i relativi dettagli teorici fare riferimento al Capitolo 3.5.

Nella versione Java di Ignite, il data streamer nativo che viene utilizzato è un'implementazione dell'interfaccia `IgniteDataStreamer`, che fornisce i metodi necessari per aggiungere i dati alla cache dalla quale lo streamer stesso è stato creato. Da sottolineare il fatto che di default il data streamer non sovrascrive i dati esistenti, pertanto se riceve delle entry che si trovano già nella cache verranno scartate, a meno che non venga esplicitamente abilitato il comportamento contrario mediante il metodo `allowOverwrite()`.

```java
// Get the data streamer reference and stream data
try (IgniteDataStreamer<Integer, String> stmr = ignite.dataStreamer("myCache")) {
    for (int i = 0; i < 100000; i++)
        stmr.addData(i, Integer.toString(i)); // stream entries
}
System.out.println("dataStreamerExample output:" + cache.get(99999));
```

Per eseguire il pre-processing dei dati prima che questi vengano aggiunti alla cache, viene messa a disposizione l'interfaccia `StreamReceiver`:

```java
try (IgniteDataStreamer<Integer, String> stmr = ignite.dataStreamer("myCache")) {
    stmr.allowOverwrite(true);
    stmr.receiver((StreamReceiver<Integer, String>) (cache, entries) ->
        entries.forEach(entry -> {
        // Do something with the entry...
        cache.put(entry.getKey(), entry.getValue());
    }));
}
```

Lo stream receiver non aggiunge automaticamente i dati così processati alla cache, per questo è necessario richiamare esplicitamente il metodo `put()`. Ignite fornisce due diverse implementazioni di stream receiver: `StreamTransformer`, che aggiorna i dati nello stream sulla base del loro precedente valore e `StreamVisitor`, che visita ciascuna coppia chiave-valore dello stream dando la possibiltà di cambiare il valore nella cache, chiamando esplicitamente anche in questo caso il metodo `put()`.

5.21 Messaging

Per i relativi dettagli teorici fare riferimento al Capitolo 3.7.

Le funzionalità di messaggistica distribuita sono disponibili attraverso l'interfaccia `IgniteMessaging`:

```
Ignite ignite = Ignition.ignite();
// Messaging instance over this cluster
IgniteMessaging msg = ignite.message();
// Messaging instance over given cluster group (in this case, remote nodes)
IgniteMessaging rmtMsg = ignite.message(ignite.cluster().forRemotes());
```

Per l'invio di messaggi si utilizza il metodo `send()` con cui specificare il messaggio e il topic. È inoltre possibile richiedere di inviare i messaggi in maniera ordinata, altrimenti di norma non lo sarebbero. Per farlo, si utilizza il metodo `sendOrdered()`, attraverso il quale si può impostare anche il parametro di timeout che specifica per quanto tempo un messaggio può rimanere in coda in attesa dei messaggi che dovrebbero essere inviati prima di esso. Allo scadere del timeout, i messaggi non ancora inviati vengono scartati.

Per poter ricevere i messaggi è invece necessario richiamare un apposito metodo di listen mediante il quale specificare il topic su cui mettersi in ascolto. Dipendentemente da come l'oggetto di messaging è stato creato, tale listener verrà associato a tutti i nodi del cluster o solo a un certo cluster group. Nello specifico, il metodo `localListen()` permette di registrare il listener solo sul nodo locale, mentre `remoteListen()` consente di registrarlo su tutti i nodi del cluster group specificato. In entrambi i casi, il listener si metterà in ascolto dei messaggi, aventi il topic desiderato, provenienti da un qualsiasi altro nodo del cluster group.

```
Ignite ignite = Ignition.ignite();
IgniteMessaging rmtMsg = ignite.message(ignite.cluster().forRemotes());
// Add listener for ordered messages on all remote nodes
rmtMsg.remoteListen("MyOrderedTopic", (nodeId, msg) -> {
    System.out.println("Received ordered message [msg=" + msg + ", from=" + nodeId + ']');

    return true; // return true to continue listening
});
// Send ordered messages to remote nodes
for (int i = 0; i < 10; i++)
    rmtMsg.sendOrdered("MyOrderedTopic", Integer.toString(i),0);
```

6 Casi d'uso e scenari di utilizzo

Come già anticipato nell'introduzione, Apache Ignite viene utilizzato da diverse aziende leader nel proprio settore, quali Microsoft, Apple, PayPal, Netflix e DreamWorks, solo per citarne alcune. Al di là di queste grandi realtà, è possbile individuare numerosi altri casi d'uso documentati che hanno potuto trarre beneficio dalle feature offerte da Ignite, in contesti industriali ma anche nell'ambito della ricerca.

6.1 Scenari aziendali

Secondo un report di Enlyft[7], sono quasi 500 le aziende che si affidano ad Ignite e nella maggior parte dei casi si tratta di compagnie di grandi dimensioni formate da decine di migliaia di impiegati e con un fatturato di oltre un miliardo di dollari. Sebbene il settore in cui viene maggiormente impiegata la tecnologia sia quello dell'IT e dei servizi, diversi casi d'uso aziendali di successo si sono riscontrati in molti altri settori, come quello delle telecomunicazioni, dei servizi finanziari, delle banche, dei trasporti, dell'industria farmaceutica e dell'*healthcare*.

Di seguito si riportano i casi d'uso più interessanti[8] [9], suddivisi in base all'area di riferimento.

Banking Nell'ambito bancario e in quello dei servizi finanziari, Ignite è stato diffusamente utilizzato per gestire la scalabilità di questi sistemi e la riduzione del carico, nonché per garantire un'elaborazione real-time a fronte di grosse moli di dati in input. Ad esempio, per effetto della pandemia, **Raiffeisen Bank** ha visto incrementare notevolmente il carico sul proprio sistema bancario a partire dai primi mesi del 2020, dal momento che gli utenti hanno potuto affidarsi esclusivamente ai canali digitali. Per questo motivo, hanno impiegato Ignite come caching layer tra il front-end (*i.e.* l'applicazione di home-banking) e il back-end responsabile delle operazioni sui conti riuscendo a ridurre il carico su quest'ultimo e ad incrementare l'affidabilità generale del sistema. Nel contesto di un più ampio processo di digital transformation, anche la banca **ING** si è affidata ad Ignite per migliorare le performance delle proprie applicazioni, considerando che queste si appoggiano ancora ad un sistema legacy di 40 anni fa, composto da mainframe che vengono ancora utilizzati da altre applicazioni, pertanto la migrazione risulterebbe complessa così come il tentativo di scalare verticalmente.

Trasporti L'aggregatore di tariffe di viaggio e di hotel **Expedia**, passando da un'architettura basata su Apache Cassandra a una basata su Ignite, è riuscita a ridurre il tempo di risposta per le query sui voli delle compagnie aeree da 3 secondi a 150 millisecondi, ottenendo un miglioramento del 95% per i milioni di ricerche che vengono fatte quotidianamente. Le feature di Ignite sfruttate maggiormente e che hanno portato a questo balzo nelle performance sono state l'uso degli oggetti binari, l'affinity colocation e le computazioni remote. Per quanto riguarda gli aspetti architetturali del cluster, invece, si è optato inizialmente per il meccanismo di discovery standard basato su TCP, per poi virare verso lo Zookeeper Discovery, implementato tramite un cluster aggiuntivo di

tre nodi, quando il numero complessivo di client e di server ha superato le cento unità. La compagnia aerea **American Airlines** nel tempo ha adottato diversi data model per memorizzare e gestire i propri dati, ma ognuno di essi presentavano svantaggi specifici che non li rendevano ottimali per il loro business: il modello chiave-valore non offriva supporto alle operazioni di join e aveva tempi di risposta troppo lenti, il modello ad oggetti garantiva poca flessibilità, il modello a grafo non risultava essere adatto alla mole di computazioni e aggragazioni previste mentre il modello colonnare portava con sé problemi di duplicazione e incosistenza. L'adozione di Ignite, integrato a Cassandra, ha permesso di superare questi problemi e migliorare così i tempi di risposta, la disponibilità dei dati, l'efficienza e la scalabilità, oltre che a fornire supporto allo standard SQL e ad offrire le garanzie delle transazioni ACID-compliant, senza dover per questo modificare il data model con cui opera Cassandra.

Healthcare Un altro campo in cui l'in-memory computing può dare un contributo fondamentale è quello della medicina di precisione e della bioinformatica. Il gruppo britannico **e-Therapeutics** si occupa della scoperta di nuovi farmaci e trattamenti per malattie biocomplesse, come il cancro e la neurodegenerazione. Nello specifico, tramite un approccio chiamato *network pharmacology*, vengono identificati e analizzati specifici gruppi di proteine associate a una certa malattia, quindi identificati più punti di intervento che, se colpiti contemporaneamente, possono interropmere questa rete di proteine. L'obiettivo è quindi trovare delle molecole che abbiano il migliore impatto complessivo sulla rete proteica e per farlo non ci si basa sul lavoro svolto in laboratorio, bensì sull'analisi computazionale delle cellule malate. Dal momento che è necessario eseguire molteplici analisi che coinvolgano diversi set di parametri e di ipotesi, l'esigenza di parallelizzare questi calcoli è evidente. Per questo motivo, l'azienda si è affida ad Ignite, implementando una piattaforma costituita da 100 nodi distribuiti su 5 commodity server. Ciò ha portato ad un incremento in velocità di circa 80 volte rispetto alla soluzione non parallelizzata, permettendo di completare in poche ore analisi che altrimenti avrebbero impiegato settimane. Un contesto simile in cui l'uso di Ignite è stato fondamentale è quello della compagnia **nference.ai**, il cui scopo principale è quello di sintetizzare tutta la conoscenza biomedica in un'unica piattaforma, cercando di integrare dati non strutturati, provenienti ad esempio da paper o articoli di ricerca, con i dati strutturati presenti nei vari database medici, contenenti risultati di test clinici o di laboratorio, e che molto spesso sono isolati e non comunicano tra di loro. La frammentazione e la replicazione offerte da Ignite, assieme alla possibilità di co-locare le computazioni presso i nodi in cui risiedono i dati, ha permesso di distribuire in maniera efficiente le computazioni, con risultati evidenti sulle performance.

6.2 Scenari di ricerca

Apache Ignite riveste un ruolo importante anche in alcuni contesti di ricerca. In qualità di tecnologia in-memory che permette di gestire le computazioni e la memorizzazione dei dati in una cache distribuita e fault tolerant, capace quindi di migliorare le performance e la scalabilità eliminando i colli di bottiglia dovuti alle operazioni di I/O del disco, è stata

sfruttata con successo in scenari in cui era richiesto di gestire dati distribuiti, eventualmente in tempo reale, oppure semplicemente per supportare task di high-performance computing.

Con il rapido crescere dell'economia globale, sono aumentate le transazioni finanziarie, di cui fanno parte anche le compravendite nei mercati capitali. Durante queste compravendite, è fondamentale tenere controllata la situazione per scongiurare potenziali azioni fraudolente e nel caso generare degli allarmi sulla base di specifiche regole di trading. Tale sistema, noto come **Trade Surveillance System** (TSS), deve quindi poter analizzare e gestire un gran numero di compravendite al secondo con una bassa latenza e un alto throughput. Bansod et al.[10] hanno messo a confronto tre diverse architetture in-memory, basate su Apache Ignite, Flink e Kafka Streams, comparando i rispettivi aspetti di fault tolerance e il contributo che il caching dà al throughput di streaming. I risultati hanno mostrato che Ignite supera le altre soluzioni per quanto riguarda il Complex Event Processing (CEP), vale a dire il processing real-time di eventi e la contestuale estrazione di informazioni da essi al fine di rilevare potenziali minacce, mentre Flink è più affidabile per quanto riguarda la fault tolerance rispetto ad Ignite. La soluzione basata su Kafka Streams risulta invece essere quella a più alta latenza a causa del suo processing basato su disco, ma quella con il miglior supporto alla fault tolerance in assoluto.

Jena et al.[11] hanno sviluppato un modello, la cui implementazione sfrutta le potenzialità del data grid distribuito di Ignite e la piattaforma di event streaming di Kafka, in grado di raccogliere dati in tempo reale da diverse sorgenti allo scopo di realizzare un sistema di allerta per la sicurezza dei conducenti e dei passeggeri. Queste informazioni in tempo reale vengono raccolte sul cloud e possono essere utilizzate ad esempio da applicazioni mobile che generino notifiche puntuali in grado di evitare incidenti, o che semplicemente informino il conducente sulle condizioni della strada, sul traffico, sulle deviazioni, sui limiti di velocità e quant'altro. Anche le forze dell'ordine possono sfruttare questi dati per creare un sistema di monitoraggio attraverso il quale analizzare i pattern di comportamento e adottare misure proattive.

Gorsky et al.[12] hanno sfruttato i vantaggi della in-memory data grid per analizzare le vulnerabilità dei sistemi energetici, cosa che richiede di considerare tutte le possibili combinazioni di guasti simultanei che possono verificarsi sugli elementi del sistema energetico.

Altri lavori, hanno invece esteso le funzionalità stesse di Ignite, al fine di migliorarne ulteriormente le performance o per impiegarlo in casi d'uso non altrimenti supportati dalla versione originale.

Sojoodi et al.[13] hanno sviluppato **Ignite-GPU**, un'estensione di Ignite che permette di sfruttare la GPU per il processing parallelo. L'architettura in-memory di Ignite, infatti, fa sì che il collo di bottiglia sulle performance si sposti dall'I/O del disco alla computazione in sé. Per questo motivo, l'unico modo per migliorare le performance su un sistema di questo tipo è quello di usare processori appropriati e veloci, come possono

essere le GPU. Queste possono elaborare i dati molto più velocemente rispetto alle CPU grazie alla loro elevata potenza di elaborazione dovuta all'architettura massivamente parallela e alla elevata larghezza di banda. Un gran numero di core semplici, piccoli ed efficienti consente alle GPU di eseguire rapidamente operazioni ripetitive e simili, con un throughput elevato, e ciò le rende particolarmente vantaggiose per l'high performance computing e per l'analisi dei dati.

Gli stessi autori hanno poi applicato questa estensione per facilitare l'esecuzione degli algoritmi genetici[14], una classe di algoritmi supportata da Ignite e che simula i processi evolutivi biologici allo scopo di trovare una soluzione ottimale quando si ha un dataset ampio e complesso. Tale algoritmo, che viene impiegato in diversi contesti applicativi industriali e legati all'intelligenza artificiale, quali il computer gaming, la robotica e il routing, ben si presta ad essere ottimizzato tramite la GPU, considerato il suo comportamento iterativo e parallelo e che spesso deve gestire grandi volumi di dati. Al contrario, sfruttare la GPU quando la dimensione dell'input è tutto sommato ridotta, può essere controproducente, dal momento che in tal caso prevarrebbe l'overhead associato al trasferimento dei dati tra la memoria centrale dello host e la memoria del device con la GPU, alla conversione dei dati nel formato appropriato e alla allocazione e de-allocazione della memoria per ogni trasferimento di dati.

Il volume sempre crescente dei dati spaziali, provenienti da diverse fonti come Google Maps, i sistemi di navigazione dei veicoli e i social network location-based, e la conseguente necessità di memorizzare, processare, interrogare e analizzare questi dati in maniera efficiente, ha portato i cosiddetti *big spatial data system* ad essere un importante campo di ricerca. Alam et al.[15], nell'ambito di uno studio di comparazione sulle performance di diversi sistemi di questo tipo, hanno ideato e sviluppato **SpatialIgnite**, una soluzione basata su Ignite, che ne rappresenta a tutti gli effetti un'estensione, in grado di gestire i dati spaziali. Il benchmark che è stato condotto ha coinvolto SpatialHadoop e GeoSpark, rispettivamente due soluzioni basate su Hadoop e Spark, ed ha evidenziato il netto vantaggio, in termini di performance e scalabilità, di SpatialIgnite rispetto agli altri due. Il motivo di questa superiorità è da ricondurre al fatto che, nel primo caso, Hadoop è un framework che si basa sull'utilizzo del disco ed è ottimizzato per l'efficienza di I/O, pertanto le performance di questo tipo di sistemi deteriorano facilmente quando si inizia a scalare. Nel secondo caso, sebbene Spark possa gestire le computazioni interamente in memoria affidandosi ad un cluster di macchine, l'overhead dovuto allo scheduling, al coordinamento della distribuzione e al movimento dei dati non la rende la soluzione più performante delle tre.

7 Tecnologie correlate

7.1 Tecnologie alternative

Tra le possibili tecnologie alternative e concorrenti a Ignite sono state individuate principalmente le seguenti:

- **Hadoop**: per quanto riguarda lo storage (file system) e la computazione distribuita
- **Spark**: in merito alla computazione distribuita in-memory
- **Kafka**: per le funzionalità di messaging e di streaming

Queste tre tecnologie, a differenza di Iginte, coprono un insieme più ridotto di funzionalità specifiche, aspetto di cui si dovrà tenere conto quando si andrà ad effettuare il confronto.

Per mettere in relazione queste tecnologie con Ignite ci si è basati sulla documentazione delle tecnologie e su paper specifici che analizzano ed effettuano un confronto, al fine di ottenere elementi di raffronto rilevanti che non sarebbe stato possibile ottenere in un ambiente limitato come quello riproducibile in una sola macchina tramite vari container, senza quindi un cluster fisico vero e proprio.

7.1.1 Apache Hadoop

Apache Hadoop è un framework open-source che permette l'elaborazione distribuita di grandi dataset su cluter di computer utilizzando un semplice modello di programmazione. Progettato per essere scalabile anche su centinaia di macchine, non si affida all'hardware per assicurare high availability, ma sulla gestione degli errori e delle failure a livello di applicazione. Hadoop è composto dai seguenti moduli principali:

- **Hadoop Distributed File System** (**HDFS**): un file system distribuito che fornisce un accesso ad alta velocità ai dati dell'applicazione.
- **Hadoop YARN**: un framework per lo scheduling dei job e per la gestione delle risorse del cluster.
- **Hadoop MapReduce**: un sistema per l'elaborazione parallela di grandi dataset basato su YARN.

Oltre alle differenze strutturali tra le due tecnologie, è interessante confrontarne l'approccio alla computazione, alla memorizzazione e all'ottimizzazione dei dati. Di seguito, vengono valutate le differenze e le somiglianze tra Ignite e Hadoop:

- Come Ignite, anche Hadoop è pensato per le computazioni distribuite di grandi dataset su cluster di computer e per scalare da singoli server a centinaia di macchine. A livello di scalabilità, entrambi garantiscono tramite l'integrazione con ZooKeeper una maggiore estensione e possibilità di gestione del cluster.

- L'unico modo per eseguire le computazioni su Hadoop è affidarsi a MapReduce e quindi all'omonimo paradigma. Questo limite impone di organizzare le computazioni in modo che passino attraverso una fase di map e una di reduce: per gli algoritmi più complessi, ciò si traduce nella necessità di prevedere più job (ciascuno formato da una coppia di task: uno di map e uno di reduce) e quindi più passate sul disco. Infatti, MapReduce non sfrutta a pieno le nuove capacità offerte dall'hardware (*i.e.* una più ampia disponibilità di memoria RAM e di processori multi-core), ma rimane fortemente incentrato sull'uso del disco. Come conseguenza, ogni risultato intermedio derivante da un job viene prima memorizzato sulla memoria di massa, per poi essere recuperato per continuare la computazione. Per questa ragione, Hadoop tende ad avere prestazioni peggiori rispetto a tecnologie più recenti come Ignite o Spark, che gestiscono l'intera computazione in memoria.

- Hadoop utilizza HDFS come file system mentre per la gestione delle risorse usa YARN. Entrambi si basano sul pattern *master-slave*, dal momento che prevedono un'entità coordinatrice che funge da master (rispettivamente il **Name Node** e il **Resource Manager**) e più slave (rispettivamente i **Data Node** e i **Node Manager**). Per quanto riguarda HDFS, il master ha il compito di mantentere l'albero del file system assieme ai metadati relativi ai file e alle cartelle, in modo tale che sappia sempre dove un certo dato è memorizzato, mentre gli slave sono resposabili della memorizzazione vera e propria e del recupero dei dati. Per quel che riguarda YARN, invece, il master costituisce l'autorità di più alto livello che arbitra le risorse tra le varie applicazioni, mentre gli slave si occupano del monitoraggio effettivo. In Ignite, tale suddivisione di ruoli non esiste e tutti i nodi si trovano dal punto di vista logico sullo stesso livello, al netto di essere nodi client o nodi server.

- I dati possono essere memorizzati in vari formati da HDFS, sia strutturati (*e.g.* chiave-valore, MapFiles) che non (*e.g.* CSV, formato testuale), ciascuno dei quali presenta vantaggi e svantaggi specifici che lo rendono adatto per un determinato ambito di utilizzo. Ignite, invece, è limitato alla sola forma strutturata chiave-valore, che cerca di ottimizzare il più possibile e di sfruttare per più scopi (*e.g.* gestione tabelle SQL-like).

- Entrambe le tecnologie in esame sfruttano il concetto di data locality, ma con Ignite è possibile avere un maggiore controllo su di esso, ad esempio nel caso in cui si debbano eseguire join distribuiti tra tabelle in modo efficiente.

- L'uso di Hadoop è più indicato per il batch processing, dal momento che HDFS è ottimizzato per l'accesso streaming ai dati, il che significa che tipicamente si vuole accedere a tutto il file e non solo ad una porzione specifica. Per questo motivo, l'enfasi è rivolta più sull'alto throughput nell'accesso ai dati, piuttosto che sulla bassa latenza nell'accesso stesso. Di contro, Ignite è più improntato ad un uso iterativo e interattivo, viste le sue caratteristiche in-memory.

7.1.2 Apache Spark

Apache Spark è un framework open-source per il calcolo distribuito che, similmente ad Ignite, utilizza primitive in-memory che gli consentono di ottenere prestazioni migliori rispetto a tecnologie più tradizionali come Hadoop. Questo accade perché i programmi possono sfruttare direttamente le risorse di memoria messe a disposizione da un cluster ed interrogarlo ripetutamente. Tali caratteristiche rendono Spark particolarmente adatto a supportare algoritmi di apprendimento automatico, ma anche query iterative o analisi dati interattive. Combina diversi modelli di processing tra cui un motore basato su MapReduce, ma che è in grado di essere fino a 100 volte più veloce rispetto alla soluzione di Hadoop. Per funzionare, Spark necessita di un gestore di cluster (come Hadoop YARN) e di un sistema di archiviazione distribuita (come Hadoop HDFS).

Le principali astrazioni di Spark sono le seguenti:

- **Resilient Distributed Dataset** (**RDD**): rappresenta un insieme di dati, distribuiti su più partizioni, che vengono memorizzati nella memoria dei nodi dei cluster, in grado di ripartire automaticamente a seguito delle failure. Gli RDD possiedono una serie di importanti caratteristiche:
  - **Immutability**: una volta creati, non possono essere modificati. Ciò garantisce l'assenza di race condition e quindi di meccanismi di locking, facilitando la parallelizzazione a fronte di un aumento di spazio occupato.
  - **Lazy evaluation**: grazie all'immutabilità, le trasformazioni sugli RDD non vengono eseguite fino a quando non sono necessarie. Per lo stesso motivo, ogni trasformazione comporta necessariamente la creazione di un nuovo RDD da quello precedente.
  - **Cacheability**: persistono in memoria, ma possono essere spostati anche su disco se necessario.
  - **Type inference**: i tipi dei dati non vengono dichiarati ma sono inferiti.
- **Direct Acyclic Graph** (**DAG**): sono dei grafi aciclici che rappresentano la sequenza di computazioni da effettuare per portare a termine le operazioni sui dati. Ogni nodo del grafo rappresenta un RDD, e ogni vertice una trasformazione da un RDD ad un altro. I DAG sono organizzati in *stage* e *task*, suddivisioni che permettono ai nodi di lavorare in parallelo quando possibile. Uno stage rappresenta l'insieme di trasformazioni che possono essere messe in pipeline ed eseguite su un singolo nodo, mentre un task rappresenta l'unità di base dello scheduling, eseguendo lo stage su una singola partizione di dati. Ogni volta che è necessario eseguire operazioni di *shuffle* (*i.e.* redistribuire i dati tra i nodi) c'è un cambio di stage.

Di seguito, per mettere a confronto Spark e Ignite, si considera un lavoro di Stan et al.[16] che analizza e compara in maniera approfondita le due tecnologie per quanto riguarda la computazione di Big Data, considerando le rispettive funzionalità, implementazioni, architetture e performance.

Nello specifico, sono stati implementati due famosi algoritmi, **Word Count** e **k-Means Clustering**, per confrontare i due motori di elaborazione in termini di utilizzo di RAM, CPU, disco e rete, al variare della dimensione del dataset e del numero di nodi, analizzando infine le performance. Il primo algoritmo, ricevuto in input un file contenente delle frasi, ha l'obiettivo di calcolare la frequenza di occorrenza di ogni parola, restituendo come risultato quella che ricorre più spesso. Il secondo, ha l'obiettivo di raggruppare dei data point simili (rappresentati da un insieme di numeri) in K cluster, scoprendo così pattern nascosti. Il principio di partizione assegna un certo punto al cluster con la media più vicina a quella attuale.

Le evidenze riscontrate sono state le seguenti:

- Nel caso del **Word Count**, il test è stato effettuato prima con uno e poi con una coppia di nodi, utilizzando file contenenti 2 o 9 milioni di parole. La misura più rilevante per questo test è stata il tempo di esecuzione impiegato dalle due tecnologie. Come si può vedere dal grafico in Figura 42, mentre con 2 milioni parole le tempistiche sono in linea, quando si passa a un file più pesante si nota una netta differenza a favore di Spark, e la differenza è ancor più evidente all'aumentare dei nodi. Un altro risultato osservabile è che a parità di nodi Spark non presenta grosse differenze nel passaggio dal file piccolo a quello più grande, mentre in Ignite la differenza è più marcata. Si può quindi concludere che Spark è in grado di gestire meglio i dataset più grandi, mentre con dataset di dimensioni più contenute le due tecnologie hanno prestazioni simili. Dal punto di vista dei consumi, dai test effet-

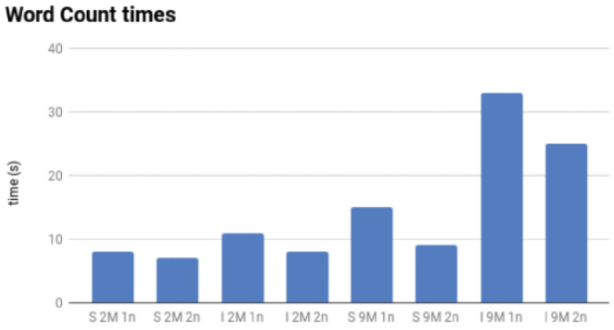

Figura 42: Tempi di esecuzione di Word Count (S = Spark, I = Ignite, n = numero nodi, M = dimensione file in milioni di parole)

tuati con due device (aventi rispettivamente 8GB e 16GB di RAM) è emerso che Apache Ignite utilizza maggiormente la CPU, ciò significa che Spark riesce ad ottenere prestazioni migliori con meno risorse computazionali. È stato notato anche che Spark non ha inviato molto lavoro al secondo nodo poiché il primo, che è anche il master, è sufficiente per il carico di lavoro assegnato, mentre Ignite condivide il carico di lavoro equamente su tutti i nodi disponibili. Per quanto riguarda l'utilizzo del disco, con Spark sono stati riscontrati due picchi di utilizzo, uno più rilevante durante la lettura iniziale dei dati e un altro nel momento di scrittura finale del

risultato. Ignite, invece, ha avuto un utilizzo medio del disco minore, con un solo picco per la scrittura del risultato. Relativamente all'utilizzo della rete, con Spark si è assistito a un picco solo alla fine, probabilmente per comunicare al master la conclusione dell'esecuzione e per il trasferimento del risultato. Per Ignite, ancora una volta, l'utilizzo è costante e ciò è spiegabile dal fatto che durante la computazione vi è un grande scambio di messaggi sulla rete, provenienti da entrambi i nodi: da una parte il server che trasmette i dati e dall'altra il client che li riceve. La presenza di tanti messaggi è il motivo principale che porta ad Ignite ad avere un tempo di esecuzione maggiore. Infine, per quanto riguarda l'utilizzo della RAM, nel caso di Spark la percentuale utilizzata è proporzionale a quella effettivamente disponibile sul device, mentre in Ignite sul device più prestante (con 16GB di RAM) si è notata una limitazione a 8GB per impostazione dei nodi. Anche questo sbilanciamento nell'utilizzo delle risorse è una delle ragioni per cui Ignite impiega più tempo di Spark per eseguire lo stesso algoritmo con la stessa quantità di dati di input.

- Nel caso del **k-Means Clustering**, le principali differenze rispetto al precedente algoritmo sono che il file di input contiene numeri anziché parole e che la dimensione è maggiore di quasi 6 volte. Nonostante ciò, come si può notare dalla Figura 43, i tempi di esecuzione rimangono simili a quelli del caso precedente, così come il pattern di comportamento, e le considerazioni fatte sopra rimangono valide anche in questo caso. Seguono lo stesso andamento anche l'utilizzo della CPU, della RAM, del disco e della rete, al netto di consumi maggiori proporzionalmente al numero di dati.

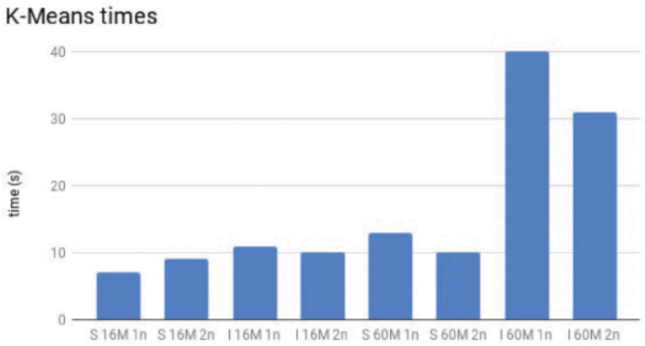

Figura 43: Tempi di esecuzione di k-Means Clusternig (S = Spark, I = Ignite, n = numero nodi, M = dimensione file in milioni di parole)

Considerando i risultati ottenuti dai test, si può concludere che Spark ha ottenuto prestazioni migliori rispetto ad Ignite. Riassumendo, è stato osservato che Ignite non gestisce bene la distribuzione dei dati sui nodi disponibili, non sfruttando tutte le risorse disponibili nel cluster. Inoltre, un altro motivo per cui Ignite è più lento di Spark, è il fatto che non bilancia bene la comunicazione tra i nodi. Un vantaggio che Ignite ha invece

su Spark è che mantiene i suoi dati nella RAM il più possibile, avendo un'interazione minima con il disco.

7.1.3 Apache Kafka

Apache Kafka è un motore di stream processing distribuito che consente di costruire pipeline di dati in real-time e applicazioni in streaming. Permette di ricevere dati da diversi tipi di sorgenti, detti *producer*, di elaborarli all'interno della sua architettura e di renderli disponibili ai riceventi, chiamati *consumer*.

La gestione dei flussi di dati non è l'unica potenzialità di Kafka, che offre una serie di funzionalità aggiuntive:

- **Storage distribuito**: consente la memorizzazione dei messaggi per prevenire la perdita dei dati e di stabilire un preciso tempo di conservazione degli stessi.
- **ETL** (Extract, Transform, Load): raccolta, trasformazione e caricamento dei dati su altri sistemi. Grazie al contributo di Kafka Connect è disponibile una serie di connettori dedicati (*e.g.* Elasticsearch, NEO4J, Cassandra).
- **Stream processing**: capacità di effettuare operazioni sui dati in tempo reale, agendo direttamente su di essi durante il flusso.

Kafka è pensato per offrire un modello di elaborazione dei dati in tempo reale, con garanzie di robustezza e semplice da usare. Oltre alla gestione dei flussi di dati, Kafka può essere utilizzato come strumento per lo scambio di messaggi asincroni in un ecosistema di microservizi, con lo scopo di velocizzare le comunicazioni tra le applicazioni e di ridurre al minimo il rischio di perdita di informazioni. Kafka agisce come un'istanza di messaggistica tra il mittente e il destinatario, offrendo soluzioni alle tipiche difficoltà che possono insorgere in questo tipo di collegamenti, come il problema della mancata memorizzazione temporanea di dati o messaggi, quando il destinatario non è disponibile (ad esempio a causa di problemi di rete).

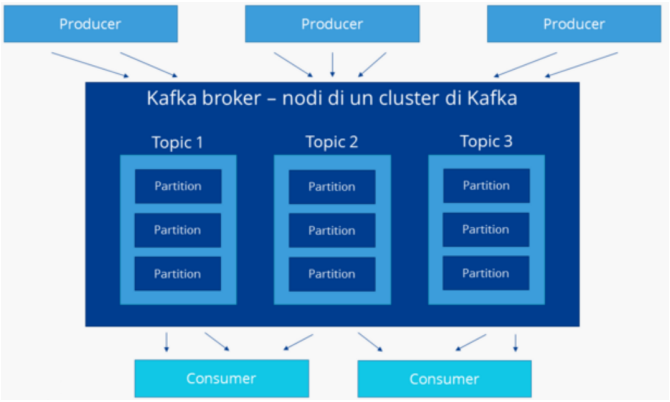

Figura 44: Architettura ed elementi principali di Kafka

L'architettura di Kafka consiste in un cluster di uno o più server. I singoli nodi, detti *broker*, salvano e categorizzano i flussi di dati in entrata in specifici *topic*. Come raffigurato nella Figura 44 della pagina precedente, ciascun broker può ospitare uno o più topic, all'interno dei quali i dati vengono suddivisi in partizioni, replicati, contrassegnati con una marca temporale e distribuiti nel cluster per essere resi disponibili ai diversi consumatori.

Le somiglianze e le differenze tra Kafka e Ignite possono essere riassunte nel modo seguente:

- Entrambe le tecnologie offrono funzionalità di streaming e di messaging con la differenza che Kafka è specializzato in questi ambiti.
- Entrambe le tecnologie si appoggiano a un cluster di nodi che possono scalare ed entrambe possono sfruttare le potenzialità offerte da Apache Zookeeper per la gestione dei nodi.
- Il funzionamento di Kafka è basato sul disco, a differenza di Ignite che è basato sulla memoria.
- Entrambe le tecnologie si basabno sul modello publish-subscribe, tramite l'utilizzo dei topic.
- Kafka, a differenza di Ignite, possiede un sistema di memorizzazione temporanea dei messaggi e dei dati verso i destinatari che sono temporaneamente disconnessi dalla rete.
- Entrambe le tecnologie offrono meccanismi di backup e di ripristino in caso di faliure delle macchine.

In generale, Kafka è la scelta più comune quando si cercano le funzionalità di messaging e di streaming sopracitate, per le quali la tecnologia è effettivamente specializzata, a differenza di Ignite che è concepito per un utilizzo diverso e per offrire funzionalità più trasversali.

7.2 GridGain

GridGain è un prodotto di classe middleware sviluppato dall'omonima società. Rappresenta la versione commerciale di Apache Ignite in grado di fornire un'interfaccia web con cui collegarsi al cluster e lanciare query SQL sui dati qui presenti.

Le caratteristiche principali sulle quali GridGain è incentrato sono:

- **Continuous Availability**: permette di eseguire degli upgrade senza intaccare la disponibilità del dato e consente di replicare i dati su più data center globalmente distribuiti.
- **Enterprise-Grade Security and Compliance**: assicura un livello di sicurezza maggiore e che segua gli standard sulla sicurezza e sulla privacy. Permette di controllare ogni movimento dei dati e degli utenti.

- **No Data Loss**: assicura che i dati vengano sempre gestiti in maniera tale da non avere perdite, creando in automatico dei punti di ripristino che possono essere usati per effettuare il recovery.

Essendo rivolto alle aziende, questo prodotto è più curato e supportato rispetto alla controparte open-source, e in base alle funzionalità desiderate è possibile scegliere tra diverse edizioni. Nella Figura 45 sono riassunte le varie versioni disponibili e le rispettive funzionalità offerte.

Capability	APACHE Ignite	GRIDGAIN Community Edition	GRIDGAIN Enterprise Edition	GRIDGAIN Ultimate Edition
Apache Ignite Capabilities →	✓	✓	✓	✓
Supported by Control Center → (separate product, not included in any Edition)	✓	✓	✓	✓
GridGain Support Services →	✓	✓	✓	✓
Production-Ready (robust testing, patches, optimizations)		✓	✓	✓
Rolling Production Upgrades →			✓	✓
Enterprise-Grade Security →			✓	✓
Data Center Replication →			✓	✓
Split-brain Protection →			✓	✓
Enterprise Extensions → (Kafka Connect, Oracle Golden Gate, etc.)			✓	✓
IBM Mainframe z/OS Support →			✓	✓
Data Compression →				✓
Full and Incremental Snapshots →				✓
Network Backups →				✓
Point-In-Time Recovery →				✓
Heterogeneous Recovery →				✓

Figura 45: Confronto tra le diverse edizioni di GridGain disponibili e delle rispettive funzionalità

Tra i diversi prodotti offerti dalla società, si annovera uno strumento molto utile dedicato al monitoraggio e alla gestione del cluster, descritto nel capitolo successivo.

7.2.1 GridGain Control Center

Control Center è uno strumento di gestione e monitoraggio per i cluster Apache Ignite. È dotato di un'interfaccia utente grafica che aiuta ad eseguire task e monitorare i cluster. Viene reso disponibile sia come SaaS che come soluzione *on-premise*.

Le sue funzionalità più importanti sono le seguenti:

- Monitoraggio dello stato del cluster attraverso dashboard personalizzabili (vedi Figura 46 pagina seguente).

- Creazione di avvisi personalizzati per tenere traccia e reagire a oltre 200 metriche relative al cluster, ai nodi e allo storage.
- Tracciamento degli update e del ribilanciamento dei dati.
- Creazione di cache, tabelle e indici e deployment del codice direttamente dall'interfaccia grafica.
- Creazione di snapshot completi del cluster.
- Tracciamento e ottimizzazione delle performance delle operazioni computazionalmente più complesse.
- Esecuzione e ottimizzazione delle query SQL, con possibilità di monitorare i comandi già in esecuzione.
- Analisi e debugging visuale delle chiamate API durante l'esecuzione sui nodi del cluster.
- Esecuzione di backup del cluster completi, incrementali e continui per consentire il ripristino di emergenza in caso di perdita o danneggiamento dei dati.

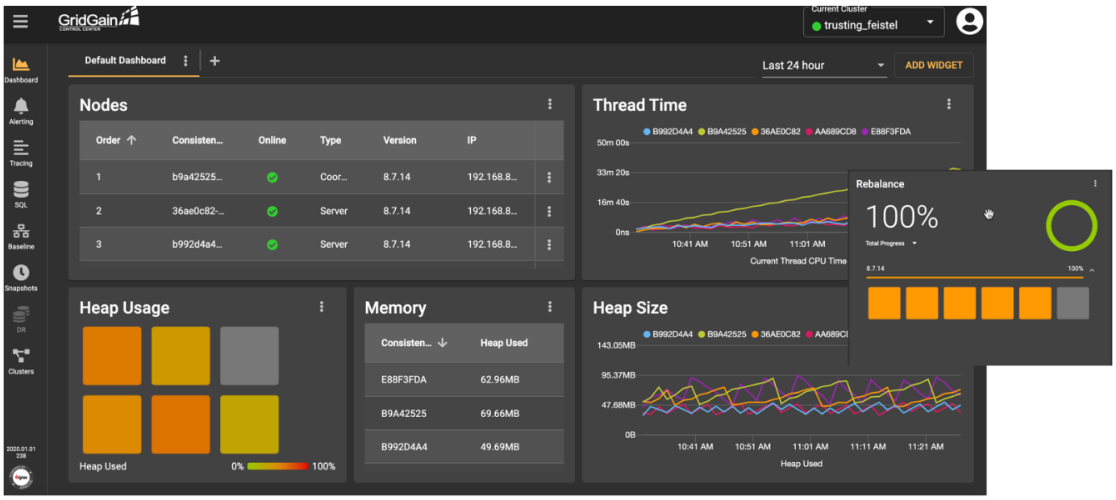

Figura 46: Esempio di dashboard monitorabile su Control Center

È possibile monitorare cluster on-premise o cluster remoti. Il meccanismo è molto semplice: basta ottenere un codice da un nodo che abbia Control Center abilitato e inserirlo nell'applicazione per registrare il cluster. A questo punto è possibile sfruttare le varie funzionalità sopracitate e osservare i dati e l'utilizzo delle risorse di calcolo del cluster e dei nodi.

8 Test

In questo capitolo vengono presentati i diversi scenari di test che sono stati implementati al fine di esplorare ed analizzare in dettaglio gli aspetti architetturali e funzionali di Ignite, assieme alle feature più importanti tra quelle presentate nelle sezioni precedenti. L'obiettivo ultimo è quello di trovare riscontro con quanto esposto nella documentazione ufficiale, verificando il comportamento della tecnologia nei casi d'uso per i quali è pensata, facendo emergere al contempo eventuali limiti o criticità.

8.1 Deployment e setup del cluster

Il progetto è stato strutturato in modo da poter propriamente usare **Gradle** come strumento di *build automation*. Nel dettaglio, si tratta di un singolo progetto multi-linguaggio in cui è stato definito un task apposito per ciascun entry-point eseguibile. In questo modo, è possibile lanciare uno certo test specificando il suo nome. A livello organizzativo, i diversi test sono stati raggruppati in base alla macro-feature in esame.

Per gestire in maniera efficace e automatizzata il deployment, in modo da renderlo riproducibile in un qualsiasi ambiente di esecuzione, si è fatto uso di **Docker**. Ciascun elemento facente parte del cluster (*i.e.* nodo server o nodo client) viene quindi eseguito su un container separato, ma in grado di comunicare con gli altri container che si trovano sulla stessa rete. Per agevolare la composizione del cluster necessario all'esecuzione di un certo test, ci si è affidati al plugin di orchestrazione **Docker Compose**, attraverso il quale è stato possibile specificare quali e quanti container istanziare, in quale modo e in quale ordine. Tali informazioni sono specificate nel seguente file `docker-compose.yaml` che definisce i due servizi che rappresentano rispettivamente i nodi server e il nodo client, assieme alla rete mediante cui si connetteranno gli uni agli altri:

```
version: "3.9"

networks:
  cluster_network:
    driver: bridge

services:
  ignite_server:
    build:
      context: .
    networks:
      - cluster_network
    environment:
      - TEST_SCENARIO=$TEST_SCENARIO_SERVER
      - TEST_SCENARIO_PARAMS=$TEST_SCENARIO_SERVER_PARAMS
    deploy:
      replicas: $SERVER_NODES
  ignite_client:
    build:
      context: .
    networks:
      - cluster_network
    environment:
      - TEST_SCENARIO=$TEST_SCENARIO_CLIENT
      - TEST_SCENARIO_PARAMS=$TEST_SCENARIO_CLIENT_PARAMS
    depends_on:
      - "ignite_server"
```

Grazie alla parametrizzazione tramite le variabili d'ambiente, con le quali si è indicato il nome del task specifico con cui eseguire il nodo server o il nodo client, gli eventuali parametri di input e il numero di nodi server da avviare, questo template può essere usato per i diversi test, avendo cura di specificare le variabili appropriate. A tale scopo, per facilitare l'esecuzione degli scenari di test, è stato fornito un file `.env` specifico per ciascuno di essi, contenente le variabili d'ambiente e i rispettivi valori (all'occorrenza modificabili) con cui si è eseguito il test. Il valore della variabile d'ambiente che veicola i parametri di input da passare al task Gradle deve essere conforme alla sintassi prevista dallo script `gradlew`, quindi seguire il pattern `-Pchiave=valore`.

Nei casi in cui sia necessaria una diversa configurazione del cluster, Compose consente di specificare un ulteriore file di configurazione, con nome `docker-compose.override.yaml`, che se presente viene letto automaticamente e come suggerisce il nome permette di andare a sovrascrivere la configurazione per i servizi specificati nel file di base ed eventualmente specificarne di nuovi.

In definitiva, per eseguire un certo test è sufficiente lanciare il seguente comando:

```
1 docker-compose --env-file <.env-file-name> -f <file-name.yaml> up
```

Nel caso sia necessario utilizzare anche il file di override, o in generale un file di configurazione aggiuntivo, basta specificarne il nome aggiungendo un'ulteriore opzione `-f`. Da sottolineare il fatto che se non si specifica niente, Compose utilizzerà in automatico sia il file con nome `docker-compose.yaml` che il file con nome `docker-compose.override.yaml`.

Una volta terminato il test, la maniera più semplice per stoppare e rimuovere la rete, tutti i container creati e gli eventuali volumi usati, ed eliminare le rispettive immagini, consiste nell'eseguire il seguente comando, avendo cura di specificare lo stesso file `.env` e gli stessi file di configurazione `.yaml` usati per lanciare il corrispondente comando di `up`:

```
1 docker-compose --env-file <.env-file-name> -f <file-name.yaml> down --rmi all -v
```

Per alcuni test si è fatto uso di **Ignite Visor**, uno strumento a linea di comando che offre delle funzionalità di base per il monitoraggio del cluster, mostrando varie statistiche relative ai nodi, alle cache e ai task eseguiti. A tale scopo, è stato predisposto un file di configurazione apposito, `docker-compose.monitoring.yaml`, che non fa altro che avviare un'istanza del tool sfruttando il modulo `ignite-visor-console` di Ignite, che offre le classi e i metodi, scritti in Scala, necessari per connettersi al servizio. Una volta avviato ed eseguito un certo test è pertanto possibile connettersi a Visor tramite il seguente comando:

```
1 docker-compose --env-file .env.monitoring -f docker-compose.monitoring.yaml run
      ignite_monitoring
```

Ciò permetterà di avviare una shell interattiva all'interno di un container che si collegherà in automatico alla rete del cluster già in esecuzione, attraverso la quale poter interagire con il tool. Affinché quest'ultimo possa connettersi correttamente al cluster, è necessario fornire come argomento il file XML di configurazione con il quale sono stati fatti partire i nodi server, il cui percorso può essere indicato nel file `.env.monitoring`.

Per ottenere invece un monitoraggio più approfondito e visuale del cluster si è fatto uso di **Control Center**, nella sua versione SaaS accessibile tramite web browser. La possibilità di costruire delle dashboard personalizzate, infatti, risulta essere molto utile per tenere sotto controllo la situazione del cluster in tempo reale. Per utilizzare il servizio, è stata definita la proprietà `clusterMonitoring` nel file `gradle.properties` del progetto che abilita il modulo `control-center-agent` di GridGain. La sua presenza fa sì che contestualmente all'avvio di ogni nodo venga stampato l'URL necessario a collegare il cluster in costruzione a Control Center e procedere quindi con il monitoraggio. Nel caso non si voglia sfruttare il servizio, basterà commentare la proprietà prevista allo scopo nel file sopracitato. A livello logico, l'interazione che avviene tra le diverse parti coinvolte può essere riassunta nella Figura 47.

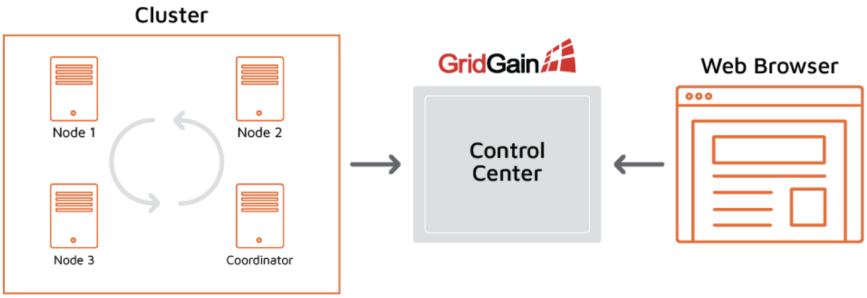

Figura 47: Schema di interazione tra Control Center, il cluster e il web browser

Immesso nel browser l'URL che compare all'avvio di un nodo (Figura 48 in alto della pagina seguente), verrà chiesto di specificare la versione di Ignite utilizzata (Figura 48 in basso a sinistra della pagina seguente) e cliccando su "CONTINUE" Control Center provvederà a cercare il cluster in questione. Una volta trovato (Figura 48 in basso a destra della pagina seguente), basterà cliccare su "ATTACH" per iniziare ad usare il servizio.

La versione di Ignite utilizzata per il progetto è la 2.14, quella più recente disponibile al momento dello sviluppo dei test. La versione di Java utilizzata per le implementazioni è la 8. La versione di Scala utilizzata per avviare il monitoraggio con Visor è la 2.12.

Gli scenari di test e le rispettive analisi sulle performance sono state effettuate su due macchine diverse, con le seguenti specifiche tecniche:

Macchina 1:

- **SO**: macOS 12.6 (architettura x86-64)
- **CPU**: Intel Core i5-7360U (dual-core, 4 core logici) - 2.3 GHz
- **RAM**: 8 GB

Macchina 2:

- **SO**: Windows 10 Pro 21H2 (architettura x86-64)
- **CPU**: AMD Rayzen 7-4700u (octa-core, 16 core logici) - 2 GHz
- **RAM**: 16 GB

Le risorse effettivamente sfruttabili sono però limitate dalla quantità delle stesse che viene specificatamente riservata a Docker. Nel caso della Macchina 1 sono quindi stati riservati 6 GB di RAM, mentre per la Macchina 2 14 GB.

```
>>> +-------------------------------------------------------------------------------------------------------+
>>> | Open the link in a browser to monitor your cluster:                                                   |
>>> | https://portal.gridgain.com/go/24236e85-37db-4741-9115-c6715eff01e8                                   |
>>> +-------------------------------------------------------------------------------------------------------+
>>> | If you are already using Control Center, you can add the cluster manually using a one-time token:     |
>>> | 24236e85-37db-4741-9115-c6715eff01e8                                                                  |
>>> |                                                                                                       |
>>> | NOTE: this token will expire in 5 minutes.                                                            |
>>> | New token can be generated with the following command: management.(sh|bat) --token                    |
>>> +-------------------------------------------------------------------------------------------------------+
>>> | For more information about connection to GridGain Control Center, please visit:                       |
>>> | https://www.gridgain.com/docs/control-center/latest/connect-ignite-cluster                            |
>>> +-------------------------------------------------------------------------------------------------------+
```

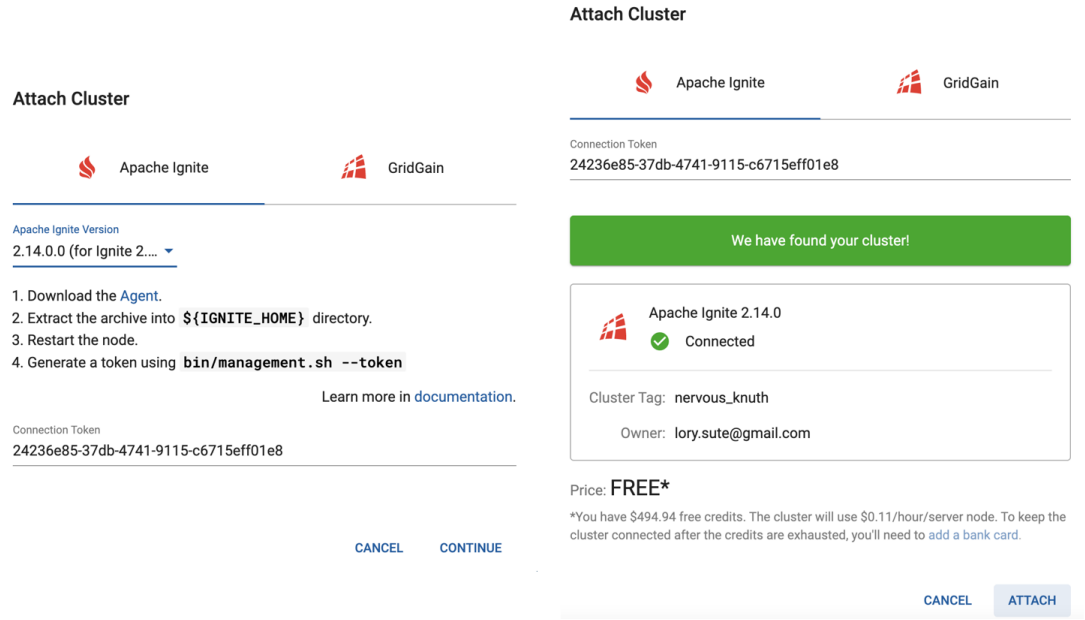

Figura 48: Esempio di connessione di un cluster Ignite a Control Center

8.2 Scenari di test

8.2.1 Partitioned mode vs. replicated mode (scenario 1.1)

Lo scopo di questo test è quello di mostrare le differenze di comportamento e di performance che si riscontrano nelle due modalità messe a disposizione da Ignite, quella partizionata e quella replicata. Nello specifico, sono state effettuate delle semplici operazioni di inserimento, lettura e aggiornamento su una cache che memorizza coppie chiave-valore di tipo intero.

Il test è stato eseguito usando la Macchina 1 e 4 nodi server, mentre per connettersi al cluster si è fatto ricorso a un thin client per sfruttare la sua proprietà di partition awareness, che consente di inviare l'operazione riguardante la cache direttamente al nodo interessato. Considerato che per sua natura il thin client non entra a far parte della topologia del cluster e non può quindi usufruire del meccanismo di discovery automatico non essendo un nodo Ignite, è necessario che esso si connetta esplicitamente mediante una connessione TCP all'indirizzo IP di un qualsiasi nodo server. Per automatizzare questa procedura si è sfruttato un volume in cui i nodi server, una volta avviati, vanno a scrivere il proprio IP (non conoscibile a priori), e da cui il thin client va a leggere. A tal fine, è stato necessario predisporre il seguente file di configurazione di override, `docker-compose.override.1.1.yaml`, con cui viene creato il volume e ritardato l'avvio del client per consentire ai server di memorizzare prima il proprio indirizzo:

```
version: "3.9"

volumes:
  address_volume:

services:
  ignite_server:
    volumes:
      - address_volume:/address_volume
  ignite_client:
    volumes_from:
      - ignite_server:r
    environment:
      - TEST_SCENARIO_CLIENT=$TEST_SCENARIO_CLIENT
      - TEST_SCENARIO_CLIENT_PARAMS=$TEST_SCENARIO_CLIENT_PARAMS
    command:
      - '/bin/sh'
      - '-c'
      - '/bin/sleep 60 && ./gradlew --no-daemon --console=plain
          $$TEST_SCENARIO_CLIENT $$TEST_SCENARIO_CLIENT_PARAMS'
    depends_on:
      - "ignite_server"
```

In Figura 49 della pagina seguente sono riportati i risultati del test. Innanzitutto, si può osservare come, in entrambe le modalità, i tempi delle letture singole superino di gran lunga quelli delle letture bulk. Tale comportamento è spiegabile dal fatto che nel primo caso vengono generati tanti messaggi quante sono le coppie memorizzate in cache (50 mila), mentre nel secondo caso un solo messaggio viene propagato in rete. Dai tempi raccolti risulta evidente come le letture siano molto efficienti in modalità replicata, dal momento che ciascun nodo mantiene la totalità dei dati, cosa che non avviene in modalità partizionata per la quale si rende necessario contattare tutti i nodi del cluster. Viceversa, le performance in scrittura risultano essere peggiori per la modalità replicata, visto che per ogni coppia deve essere creata una copia di backup in tutti i restanti nodi.

```
ignite_test_scenarios-ignite_server-1  | MODE: partitioned
ignite_test_scenarios-ignite_server-1  | >>>    Server node IP address successfully stored in the volume.
ignite_test_scenarios-ignite_server-2  | MODE: partitioned
ignite_test_scenarios-ignite_server-2  | >>>    Server node IP address successfully stored in the volume.
ignite_test_scenarios-ignite_server-3  | MODE: partitioned
ignite_test_scenarios-ignite_server-3  | >>>    Server node IP address successfully stored in the volume.
ignite_test_scenarios-ignite_server-4  | MODE: partitioned
ignite_test_scenarios-ignite_server-4  | >>>    Server node IP address successfully stored in the volume.
ignite_test_scenarios-ignite_client-1  | Inserting time: 78763 ms
ignite_test_scenarios-ignite_client-1  | Reading time (bulk): 4467 ms
ignite_test_scenarios-ignite_client-1  | Reading time: 117712 ms
ignite_test_scenarios-ignite_client-1  | Updating time: 71759 ms
```

```
ignite_test_scenarios-ignite_server-1  | MODE: replicated
ignite_test_scenarios-ignite_server-1  | >>>    Server node IP address successfully stored in the volume.
ignite_test_scenarios-ignite_server-2  | MODE: replicated
ignite_test_scenarios-ignite_server-2  | >>>    Server node IP address successfully stored in the volume.
ignite_test_scenarios-ignite_server-3  | MODE: replicated
ignite_test_scenarios-ignite_server-3  | >>>    Server node IP address successfully stored in the volume.
ignite_test_scenarios-ignite_server-4  | MODE: replicated
ignite_test_scenarios-ignite_server-4  | >>>    Server node IP address successfully stored in the volume.
ignite_test_scenarios-ignite_client-1  | Inserting time: 97008 ms
ignite_test_scenarios-ignite_client-1  | Reading time (bulk): 604 ms
ignite_test_scenarios-ignite_client-1  | Reading time: 30452 ms
ignite_test_scenarios-ignite_client-1  | Updating time: 66336 ms
```

Figura 49: Output dello scenario di test 1.1 - modalità partitioned (a sinistra) vs. modalità replicated (a destra)

Per avere conferma dell'effettiva organizzazione delle cache nelle due modalità si è fatto uso di Visor, nello specifico del comando `cache -a` tramite il quale è possibile vedere delle informazioni di base. In modalità partitioned (Figura 50) si vede come le entry vengano automaticamente bilanciate tra i nodi server a disposizione, nella memoria off-heap. In modalità replicated (Figura 51 pagina seguente), invece, oltre alla copia primaria che ciascun nodo memorizza è presente una copia di backup per ciascuna coppia salvata nei restanti nodi.

```
visor> cache -a
cache -a
Time of the snapshot: 2023-03-16 15:48:45
+=============================================================================================================================================================+
|  Name(@)   |    Mode     | Nodes | Total entries (Heap / Off-heap) | Primary entries (Heap / Off-heap) |   Hits    |  Misses   |   Reads   |  Writes   |
+=============================================================================================================================================================+
| cache(@c0) | PARTITIONED | 4     | 50000 (0 / 50000)               | min: 12211 (0 / 12211)            | min: 0    | min: 0    | min: 0    | min: 0    |
|            |             |       |                                 | avg: 12500.00 (0.00 / 12500.00)   | avg: 0.00 | avg: 0.00 | avg: 0.00 | avg: 0.00 |
|            |             |       |                                 | max: 12836 (0 / 12836)            | max: 0    | max: 0    | max: 0    | max: 0    |
+-------------------------------------------------------------------------------------------------------------------------------------------------------------+

Cache 'cache(@c0)':
+-------------------------------------------------------------+
| Name(@)                          | cache(@c0)               |
| Total entries (Heap / Off-heap)  | 50000 (0 / 50000)        |
| Nodes                            | 4                        |
| Total size Min/Avg/Max           | 12211 / 12500.00 / 12836 |
|   Heap size Min/Avg/Max          | 0 / 0.00 / 0             |
|   Off-heap size Min/Avg/Max      | 12211 / 12500.00 / 12836 |
+-------------------------------------------------------------+

Nodes for: cache(@c0)
+=============================================================================================================+
|      Node ID8(@), IP      | CPUs | Heap Used | CPU Load |   Up Time    |    Size (Primary / Backup)    | Hi/Mi/Rd/Wr |
+=============================================================================================================+
| 3BD515A5(@n1), 172.24.0.2 | 4    | 8.24 %    | 0.90 %   | 00:06:03.464 | Total: 12836 (12836 / 0)      | Hi: 0       |
|                           |      |           |          |              |   Heap: 0 (0 / <n/a>)         | Mi: 0       |
|                           |      |           |          |              |   Off-Heap: 12836 (12836 / 0) | Rd: 0       |
|                           |      |           |          |              |   Off-Heap Memory: <n/a>      | Wr: 0       |
+---------------------------+------+-----------+----------+--------------+-------------------------------+-------------+
| 94EFBCD1(@n0), 172.24.0.5 | 4    | 6.48 %    | 1.00 %   | 00:06:03.710 | Total: 12303 (12303 / 0)      | Hi: 0       |
|                           |      |           |          |              |   Heap: 0 (0 / <n/a>)         | Mi: 0       |
|                           |      |           |          |              |   Off-Heap: 12303 (12303 / 0) | Rd: 0       |
|                           |      |           |          |              |   Off-Heap Memory: <n/a>      | Wr: 0       |
+---------------------------+------+-----------+----------+--------------+-------------------------------+-------------+
| FBED9F25(@n3), 172.24.0.3 | 4    | 5.63 %    | 0.90 %   | 00:06:04.469 | Total: 12650 (12650 / 0)      | Hi: 0       |
|                           |      |           |          |              |   Heap: 0 (0 / <n/a>)         | Mi: 0       |
|                           |      |           |          |              |   Off-Heap: 12650 (12650 / 0) | Rd: 0       |
|                           |      |           |          |              |   Off-Heap Memory: <n/a>      | Wr: 0       |
+---------------------------+------+-----------+----------+--------------+-------------------------------+-------------+
| 61E9B1B0(@n2), 172.24.0.4 | 4    | 5.45 %    | 1.23 %   | 00:06:04.462 | Total: 12211 (12211 / 0)      | Hi: 0       |
|                           |      |           |          |              |   Heap: 0 (0 / <n/a>)         | Mi: 0       |
|                           |      |           |          |              |   Off-Heap: 12211 (12211 / 0) | Rd: 0       |
|                           |      |           |          |              |   Off-Heap Memory: <n/a>      | Wr: 0       |
+-------------------------------------------------------------------------------------------------------------+
'Hi' - Number of cache hits.
'Mi' - Number of cache misses.
'Rd' - number of cache reads.
'Wr' - Number of cache writes.

Aggregated queries metrics:
  Minimum execution time: 00:00:00.000
  Maximum execution time: 00:00:00.000
  Average execution time: 00:00:00.000
  Total number of executions: 0
  Total number of failures:   0
```

Figura 50: Monitoraggio tramite Visor - modalità partitioned

```
visor> cache -a
cache -a
Time of the snapshot: 2023-03-16 15:30:29
+=======================================================================================================================================================+
|  Name(@)   |    Mode    | Nodes | Total entries (Heap / Off-heap) | Primary entries (Heap / Off-heap) |    Hits    |   Misses   |   Reads    |   Writes   |
+=======================================================================================================================================================+
| cache(@c0) | REPLICATED | 4     | 50000 (0 / 50000)               | min: 10455 (0 / 10455)            | min: 0     | min: 0     | min: 0     | min: 0     |
|            |            |       |                                 | avg: 12500.00 (0.00 / 12500.00)   | avg: 0.00  | avg: 0.00  | avg: 0.00  | avg: 0.00  |
|            |            |       |                                 | max: 14061 (0 / 14061)            | max: 0     | max: 0     | max: 0     | max: 0     |
+-------------------------------------------------------------------------------------------------------------------------------------------------------+

Cache 'cache(@c0)':
+-----------------------------------------------------------+
| Name(@)                         | cache(@c0)              |
| Total entries (Heap / Off-heap) | 50000 (0 / 50000)       |
| Nodes                           | 4                       |
| Total size Min/Avg/Max          | 10455 / 12500.00 / 14061 |
|   Heap size Min/Avg/Max         | 0 / 0.00 / 0            |
|   Off-heap size Min/Avg/Max     | 10455 / 12500.00 / 14061 |
+-----------------------------------------------------------+

Nodes for: cache(@c0)
+=====================================================================================================================+
|      Node ID8(@), IP      | CPUs | Heap Used | CPU Load |   Up Time    |       Size (Primary / Backup)       | Hi/Mi/Rd/Wr |
+=====================================================================================================================+
| D0152834(@n2), 172.23.0.4 | 4    | 5.42 %    | 1.00 %   | 00:19:54.306 | Total: 50000 (13574 / 36426)         | Hi: 0       |
|                           |      |           |          |              |   Heap: 0 (0 / <n/a>)                | Mi: 0       |
|                           |      |           |          |              |   Off-Heap: 50000 (13574 / 36426)    | Rd: 0       |
|                           |      |           |          |              |   Off-Heap Memory: <n/a>             | Wr: 0       |
+---------------------------+------+-----------+----------+--------------+--------------------------------------+-------------+
| 62E7A502(@n0), 172.23.0.2 | 4    | 6.52 %    | 5.10 %   | 00:19:58.014 | Total: 50000 (10455 / 39545)         | Hi: 0       |
|                           |      |           |          |              |   Heap: 0 (0 / <n/a>)                | Mi: 0       |
|                           |      |           |          |              |   Off-Heap: 50000 (10455 / 39545)    | Rd: 0       |
|                           |      |           |          |              |   Off-Heap Memory: <n/a>             | Wr: 0       |
+---------------------------+------+-----------+----------+--------------+--------------------------------------+-------------+
| 6C34D0A9(@n1), 172.23.0.3 | 4    | 5.61 %    | 1.60 %   | 00:19:56.288 | Total: 50000 (14061 / 35939)         | Hi: 0       |
|                           |      |           |          |              |   Heap: 0 (0 / <n/a>)                | Mi: 0       |
|                           |      |           |          |              |   Off-Heap: 50000 (14061 / 35939)    | Rd: 0       |
|                           |      |           |          |              |   Off-Heap Memory: <n/a>             | Wr: 0       |
+---------------------------+------+-----------+----------+--------------+--------------------------------------+-------------+
| 27E3DEE8(@n3), 172.23.0.5 | 4    | 15.93 %   | 0.90 %   | 00:19:52.727 | Total: 50000 (11910 / 38090)         | Hi: 0       |
|                           |      |           |          |              |   Heap: 0 (0 / <n/a>)                | Mi: 0       |
|                           |      |           |          |              |   Off-Heap: 50000 (11910 / 38090)    | Rd: 0       |
|                           |      |           |          |              |   Off-Heap Memory: <n/a>             | Wr: 0       |
+---------------------------------------------------------------------------------------------------------------------+
'Hi' - Number of cache hits.
'Mi' - Number of cache misses.
'Rd' - number of cache reads.
'Wr' - Number of cache writes.

Aggregated queries metrics:
  Minimum execution time: 00:00:00.000
  Maximum execution time: 00:00:00.000
  Average execution time: 00:00:00.000
  Total number of executions: 0
  Total number of failures:   0
```

Figura 51: Monitoraggio tramite Visor - modalità replicated

8.2.2 On-heap caching vs. off-heap caching (scenario 1.2)

Similmente allo scenario precedente, vengono in questo caso effettuate delle operazioni di inserimento, lettura e aggiornamento dei dati in cache, mettendo a confronto la modalità di caching già impiegata precedentemente e attiva di default, vale a dire quella off-heap, con quella on-heap. Dal punto di vista teorico, quest'ultima può garantire delle performance in lettura maggiori rispetto allo scenario tradizionale, dal momento che una copia della cache off-heap (che rimane comunque presente anche in questa modalità) viene mantenuta nella memoria heap della JVM in cui è in esecuzione il nodo. Così facendo, vengono meno tutti i tempi necessari alla serializzazione/deserializzazione dei dati.

Il test è stato eseguito usando la Macchina 2 e 1 nodo server che si occupa di eseguire tutte le operazioni sulla cache, simulando uno scenario in cui un nodo server ha necessità di accedere ai dati che memorizza, ad esempio nel contesto di una più ampia computazione distribuita. In questo caso, il nodo client non fa niente.

Nella tabella in Figura 52 della pagina seguente sono riportati i tempi registrati per le varie operazioni nelle due modalità, al crescere del numero di entry memorizzate nella cache, i cui valori sono in questo caso degli oggetti complessi della classe `Person`.

Nel grafico in Figura 52 è possibile analizzare più nel dettaglio l'andamento dei tempi di lettura al variare del numero di dati. Nello specifico, per quantità di entry relativamente piccole si riscontra un'effettiva differenza nelle tempistiche a favore della modalità on-heap, che inizialmente impiega circa la metà rispetto alla controparte off-heap. Tuttavia, tale forbice si assottiglia man mano che aumentano i dati, con la modalità off-heap che prima eguaglia e poi supera, migliorandoli, i tempi di quella on-heap. Tale comportamento può essere ricondotto all'overhead associato al garbage collector, la cui attività aumenta all'aumentare della dimensione della cache e quindi della memoria heap utilizzata.

# entries	cache operation	OFF-HEAP	ON-HEAP
1k	PUT	283	298
	GET	115	66
	UPDATE	99	116
5k	PUT	642	823
	GET	168	115
	UPDATE	108	158
10k	PUT	809	825
	GET	228	174
	UPDATE	169	194
100k	PUT	1750	1775
	GET	560	544
	UPDATE	402	720
1M	PUT	7321	10211
	GET	3879	3829
	UPDATE	3879	6151
2M	PUT	11593	22393
	GET	8801	13124
	UPDATE	8327	13509

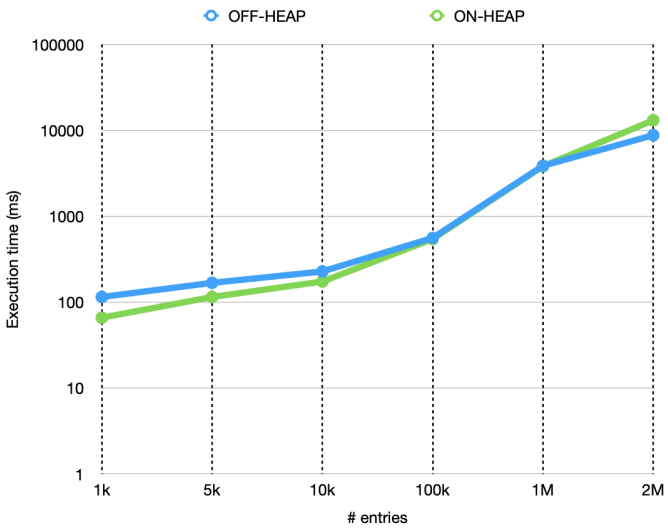

Figura 52: Tabella con i tempi (in millisecondi) delle varie operazioni sulla cache nelle due modalità (a sinistra) e grafico con l'andamento dei tempi di lettura nelle due modalità al variare del numero di dati (a destra)

Per quanto riguarda le tempistiche delle operazioni di inserimento e aggiornamento, si può notare che la modalità on-heap impiega sempre di più rispetto alla modalità off-heap. Tale evidenza può trovare spiegazione nel fatto che in modalità on-heap devono essere scritte/aggiornate il doppio delle entry. Sfruttando Visor è possibile verificare effettivamente che in modalità on-heap i dati, oltre ad essere memorizzati nella cache off-heap come al solito, vengono interamente duplicati nello heap (Figura 53 pagina seguente).

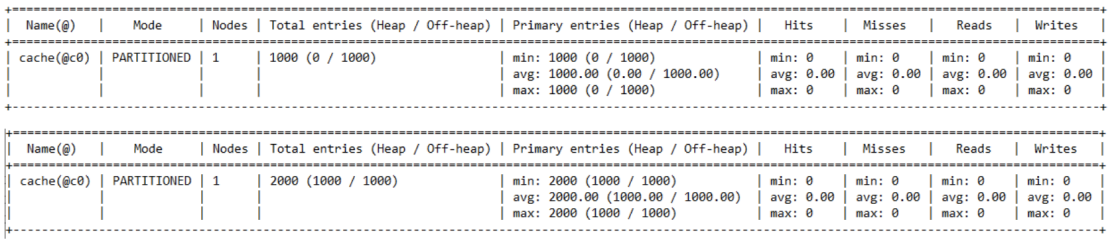

Figura 53: Monitoraggio tramite Visor - modalità off-heap (sopra) vs. modalità on-heap (sotto)

8.2.3 Binary mode vs. no-binary mode (scenario 1.3)

Con questo test si vogliono analizzare i vantaggi e gli svantaggi delle operazioni eseguite sulle cache nel caso in cui si attivi la modalità di lavoro binary rispetto a quella classica.

Di default, Ignite deserializza tutto l'oggetto che viene recuperato dalla cache in caso di operazioni di get. Con la modalità binary si può invece andare a lavorare direttamente con gli elementi salvati sulla cache senza il bisogno di conoscere la classe dell'oggetto. Questo è possibile grazie all'interfaccia `BinaryObject` che mette a disposizione il metodo `toBuilder()` per ottenere un builder di oggetti con la stessa struttura di quello su cui è stato chiamato il metodo.

Il test è strutturato in due fasi per ogni modalità, una di inserimento dati nella cache e una di update che comprende la lettura, la modifica e la riscrittura del record letto. Per la modalità classica, non c'è nulla di diverso rispetto alle tradizionali operazioni di put e di get. Per la modalità binaria, invece, nella fase di inserimento viene eseguita la put su una cache su cui viene chiamato il metodo `withKeepBinary()`, e il dato da inserire viene generato tramite un builder di `BinaryObject`. Per la fase di update, viene letto l'oggetto binario e partendo da questo si ottiene un builder che a sua volta genererà il nuovo oggetto che sostituirà quello appena letto (i `BinaryObject` sono immutabili).

Per verificare le prestazioni delle due modalità, vengono registrate e confrontate le tempistiche delle rispettive due fasi. In Figura 54, Figura 55 e Figura 56 della pagina seguente sono riportate le tempistiche ottenute utilizzando rispettivamente 10 mila, 100 mila e 1 milione di record, con 5 nodi server avviati nella Macchina 2. I risultati ottenuti, essendo molto vicini in termini di prestazioni e tempi, non sono stati molto determinanti. Ripetendo la misurazione dei tempi, è possibile che si registrino ogni volta risultati diversi, per questo motivo sono stati qui riportati quelli che più rispecchiano la media dei risultati ottenuti in diverse esecuzioni.

Dai tempi mostrati si osserva che la versione che non utilizza gli oggetti binari sembra avere tempi generalmente migliori. Dal punto di vista teorico, la differenza delle due modalità dovrebbe risiedere nell'utilizzo di classi condivise tra i nodi per la versione non binaria e nel non dover deserializzare gli oggetti ottenuti e inseriti nelle cache per la versione binaria. Nonostante questo, la versione binaria ha dato un risultato leggermente peggiore. Considerando la poca differenza nei tempi si può asserire che piuttosto che per ragioni legate alle prestazioni, la scelta di una o dell'altra modalità dovrebbe essere

fatta in base al cluster, ai dati da memorizzare e alle specifiche richieste.

```
ignite_test_scenarios-ignite_client-1  | Binary PUT operations time: 12783 ms
ignite_test_scenarios-ignite_client-1  | Binary GET operations time: 8019 ms
ignite_test_scenarios-ignite_client-1  | Binary UPDATE operations time: 13848 ms
ignite_test_scenarios-ignite_client-1  | Non Binary PUT operations time: 6719 ms
ignite_test_scenarios-ignite_client-1  | Non Binary GET operations time: 5221 ms
ignite_test_scenarios-ignite_client-1  | Non Binary UPDATE operations time: 11199 ms
```

Figura 54: Output dello scenario di test 1.3 - 10.000 record

```
ignite_test_scenarios-ignite_client-1  | Binary PUT operations time: 12783 ms
ignite_test_scenarios-ignite_client-1  | Binary GET operations time: 8019 ms
ignite_test_scenarios-ignite_client-1  | Binary UPDATE operations time: 13848 ms
ignite_test_scenarios-ignite_client-1  | Non Binary PUT operations time: 6719 ms
ignite_test_scenarios-ignite_client-1  | Non Binary GET operations time: 5221 ms
ignite_test_scenarios-ignite_client-1  | Non Binary UPDATE operations time: 11199 ms
```

Figura 55: Output dello scenario di test 1.3 - 100.000 record

```
ignite_test_scenarios-ignite_client-1  | Binary PUT operations time: 406156 ms
ignite_test_scenarios-ignite_client-1  | Binary GET operations time: 367410 ms
ignite_test_scenarios-ignite_client-1  | Binary UPDATE operations time: 809037 ms
ignite_test_scenarios-ignite_client-1  | Non Binary PUT operations time: 410028 ms
ignite_test_scenarios-ignite_client-1  | Non Binary GET operations time: 378006 ms
ignite_test_scenarios-ignite_client-1  | Non Binary UPDATE operations time: 802644 ms
```

Figura 56: Output dello scenario di test 1.3 - 1.000.000 record

8.2.4 Cache group (scenario 1.4)

In questo test si andrà a testare il funzionamento dei cache group tramite la simulazione di un cluster composto da molteplici nodi server, i quali memorizzeranno un grande numero di cache. Nel primo caso verranno usati i cache group e nel secondo si procederà senza. Quello che si cerca di evidenziare è la differenza di spazio dello heap occupato dalle cache nelle due versioni di questo test. Sfruttando i cache group in un cluster in modalità persistente, si otterrà una significativa riduzione dello spazio richiesto per il mantenimento in memoria delle partition map. In questo caso, infatti, le cache appartenenti allo stesso cache group condivideranno la stessa partition map, diversamente dalla situazione standard nella quale ciascuna cache ha la propria partition map. Per ottenere questi risultati sono state attivate la persistenza sul cluster e le relative metriche, in modo da monitorare lo spazio occupato nello heap.

In entrambi i casi il test è stato eseguito con 3 nodi server sulla Macchina 2, creando 70 cache e inserendo 1000 record in ciascuna di esse. Nel primo caso, quello in cui è presente il cache group, è stata effettivamente riscontrata in maniera evidente una riduzione dello spazio occupato nello heap. Questa differenza si può osservare nell'output ottenuto dalle metriche di persistenza riportato in Figura 57 e in Figura 58 della pagina seguente. Da notare come nel primo test solamente un nodo segnali una occupazione di memoria nello heap, questo perché, come detto sopra, è presente una sola partition map per tutte le cache. La somma dello spazio occupato ottenuto nell'output del primo test è pari a 35 MB, nel secondo invece si arriva ad 2493 MB.

```
ignite_test_scenarios-ignite_server-1 | [20:57:36] Data storage metrics for local node (to disable set 'metricsLogFrequency' to 0)
ignite_test_scenarios-ignite_server-1 | [20:57:36]     ^-- Off-heap memory [used=0MB, free=100%, allocated=2998MB]
ignite_test_scenarios-ignite_server-1 | [20:57:36]     ^-- Page memory [pages=25]
ignite_test_scenarios-ignite_server-1 | [20:57:36]     ^-- Ignite persistence [used=0MB]
ignite_test_scenarios-ignite_server-2 | [20:57:32] Data storage metrics for local node (to disable set 'metricsLogFrequency' to 0)
ignite_test_scenarios-ignite_server-2 | [20:57:32]     ^-- Off-heap memory [used=35MB, free=98.88%, allocated=2998MB]
ignite_test_scenarios-ignite_server-2 | [20:57:32]     ^-- Page memory [pages=9025]
ignite_test_scenarios-ignite_server-2 | [20:57:32]     ^-- Ignite persistence [used=35MB]
ignite_test_scenarios-ignite_server-3 | [20:57:37] Data storage metrics for local node (to disable set 'metricsLogFrequency' to 0)
ignite_test_scenarios-ignite_server-3 | [20:57:37]     ^-- Off-heap memory [used=0MB, free=100%, allocated=2998MB]
ignite_test_scenarios-ignite_server-3 | [20:57:37]     ^-- Page memory [pages=25]
ignite_test_scenarios-ignite_server-3 | [20:57:37]     ^-- Ignite persistence [used=0MB]
```

Figura 57: Output dello scenario di test 1.4 - con cache group

```
ignite_test_scenarios-ignite_server-1 | [20:51:49] Data storage metrics for local node (to disable set 'metricsLogFrequency' to 0)
ignite_test_scenarios-ignite_server-1 | [20:51:49]     ^-- Off-heap memory [used=862MB, free=73.02%, allocated=2998MB]
ignite_test_scenarios-ignite_server-1 | [20:51:49]     ^-- Page memory [pages=218359]
ignite_test_scenarios-ignite_server-1 | [20:51:49]     ^-- Ignite persistence [used=852MB]
ignite_test_scenarios-ignite_server-2 | [20:51:48] Data storage metrics for local node (to disable set 'metricsLogFrequency' to 0)
ignite_test_scenarios-ignite_server-2 | [20:51:48]     ^-- Off-heap memory [used=808MB, free=74.73%, allocated=2998MB]
ignite_test_scenarios-ignite_server-2 | [20:51:48]     ^-- Page memory [pages=204499]
ignite_test_scenarios-ignite_server-2 | [20:51:48]     ^-- Ignite persistence [used=798MB]
ignite_test_scenarios-ignite_server-3 | [20:51:49] Data storage metrics for local node (to disable set 'metricsLogFrequency' to 0)
ignite_test_scenarios-ignite_server-3 | [20:51:49]     ^-- Off-heap memory [used=823MB, free=74.27%, allocated=2998MB]
ignite_test_scenarios-ignite_server-3 | [20:51:49]     ^-- Page memory [pages=208279]
ignite_test_scenarios-ignite_server-3 | [20:51:49]     ^-- Ignite persistence [used=813MB]
```

Figura 58: Output dello scenario di test 1.4 - senza cache group

8.2.5 Data rebalancing (scenario 1.5)

Per tale test è stato usato Control Center, configurando un'apposita dashboard con dei widget selezionati per mostrare risultati e informazioni utili (Figura 59). Per le istruzioni su come accedere e collegare il cluster al servizio fare riferimento al Capitolo 8.1.

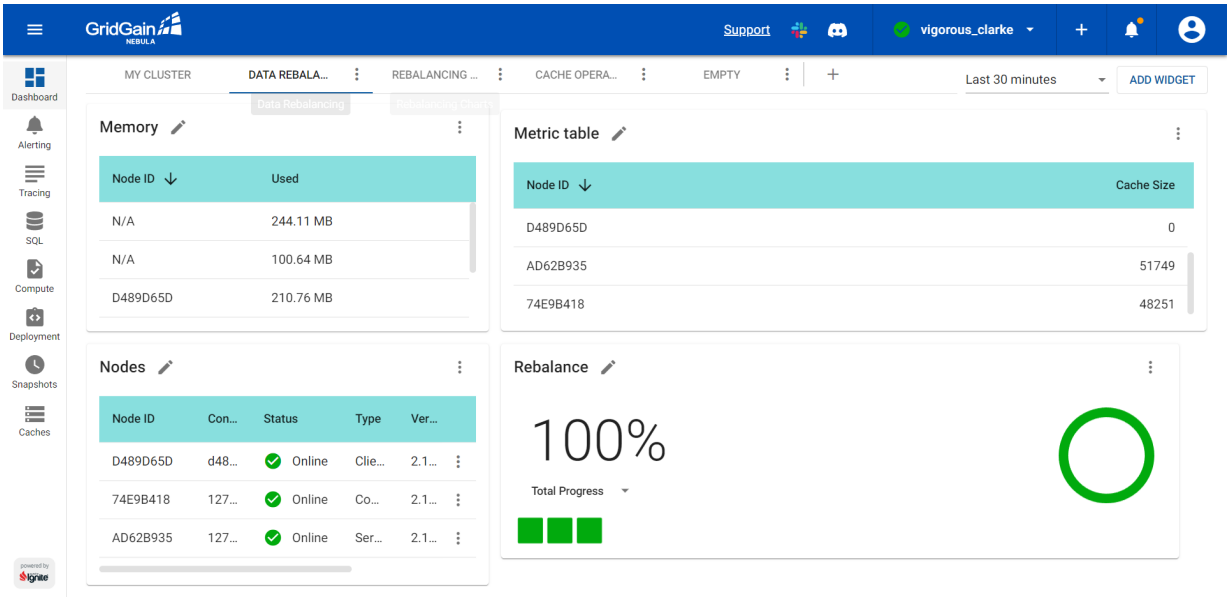

Figura 59: Dashboard configurata per la visualizzazione di diverse metriche su Control Center

Sfruttando questo strumento di monitoring, che permette di ottenere una grande varietà di informazioni dalle cache create, l'obiettivo è di osservare il rebalancing dei dati conseguente a diversi eventi che causano un cambiamento nella baseline topology. Nello specifico, vengono fatti partire 2 nodi server e viene creata una cache a cui vengono aggiunti 2000 record. La Figura 60 mostra l'intercettazione della spartizione dei dati tra i nodi.

Node ID ↓	Cache Size
D489D65D	0
332C1FEA	974
131235E4	1026

Figura 60: Ripartizione dei record nei 2 nodi server dopo l'inserimento iniziale dei dati in cache

Successivamente, aggiungendo altri 2 server (che compaiono come seconda e terza riga nella Figura 61), viene scatenato il rebalancing dei dati, come si può osservare dalla Figura 62.

Node ID ↓	Cache Size
D489D65D	0
AD62B935	0
74E9B418	0
332C1FEA	974
131235E4	1026

Figura 61: Inserimento di ulteriori 2 nodi server

Node ID ↓	Cache Size
D489D65D	0
AD62B935	504
74E9B418	484
332C1FEA	508
131235E4	504

Figura 62: Situazione dopo il rebalancing innescato dall'aggiunta dei nodi

Infine, dopo l'aggiunta di ulteriori 98 mila record, operazione che inserisce i dati nelle corrette partizioni dei 4 server (Figura 63 pagina seguente), si simula il crash di 2 nodi per osservare, in Figura 64 e in Figura 65 della pagina seguente, la distribuzione dei dati prima e dopo quest'ultimo evento di rebalancing.

Si fa notare che al fine di osservare l'effettiva sequenza di rebalancing derivante dai diversi eventi sopracitati, sono state inserite appositamente delle sleep.

In questo test viene anche mostrato il funzionamento degli eventi relativi alle operazioni della cache, impostando un listener nei nodi remoti che intercetta gli eventi di put. È possibile osservare l'effettiva ricezione degli eventi dall'output dei server. Il listener viene creato prima dell'aggiunta dei primi 2000 record, dopo che questi sono stati inseriti il listener viene fermato, interrompendo così la ricezione degli eventi e le relative stampe.

Node ID ↓	Cache Size
D489D65D	0
AD62B935	24997
74E9B418	24226
332C1FEA	25579
131235E4	25198

Figura 63: Distribuzione dei nuovi dati nei 4 nodi server

Node ID ↓	Cache Size
N/A	25198
N/A	25579
D489D65D	0
AD62B935	24997
74E9B418	75003

Figura 64: Situazione a seguito della rimozione di 2 nodi server, subito prima del rebalancing

Node ID ↓	Cache Size
N/A	25198
N/A	25579
D489D65D	0
AD62B935	51749
74E9B418	48251

Figura 65: Situazione a seguito della rimozione di 2 nodi server, dopo il rebalancing

8.2.6 Partition loss (scenario 1.6)

In questo test viene creata una cache che tramite l'impostazione di alcuni parametri entrerà in uno stato di partition loss. Per arrivare a tale situazione, è necessario terminare un numero di nodi server maggiore del numero di backup della cache. Nello specifico, l'esecuzione qui di seguito analizzata ha coinvolto 4 nodi server, 1 backup e la chiusura di 3 nodi.

Il test si suddivide in 4 fasi: creazione dei nodi, inserimento dei dati, lettura con eventuale perdita di dati e aggiornamento. Con le ultime due fasi è possibile osservare le differenze di comportamento, in caso di perdita di dati, in base alle diverse policy impostabili sulla cache.

Inizialmente, tutti i dati sono presenti in cache (Figura 66). Quando si effettuano delle get in caso di partition loss policy impostata ad `IGNORE` (Figura 67), i record persi vengono considerati come se fossero vuoti/non presenti e l'utente non viene notificato in nessuna maniera in caso di scrittura.

```
ignite_test_scenarios-ignite_client-1  | Before node close.
ignite_test_scenarios-ignite_client-1  | 0
ignite_test_scenarios-ignite_client-1  | 1
ignite_test_scenarios-ignite_client-1  | 2
ignite_test_scenarios-ignite_client-1  | 3
ignite_test_scenarios-ignite_client-1  | 4
ignite_test_scenarios-ignite_client-1  | 5
ignite_test_scenarios-ignite_client-1  | 6
ignite_test_scenarios-ignite_client-1  | 7
ignite_test_scenarios-ignite_client-1  | 8
ignite_test_scenarios-ignite_client-1  | 9
ignite_test_scenarios-ignite_client-1  | Closing..
```

Figura 66: Inserimento e lettura di 10 record nella cache

```
ignite_test_scenarios-ignite_client-1  | After node close.
ignite_test_scenarios-ignite_client-1  | 0
ignite_test_scenarios-ignite_client-1  | 1
ignite_test_scenarios-ignite_client-1  | 2
ignite_test_scenarios-ignite_client-1  | null
ignite_test_scenarios-ignite_client-1  | null
ignite_test_scenarios-ignite_client-1  | 5
ignite_test_scenarios-ignite_client-1  | 6
ignite_test_scenarios-ignite_client-1  | 7
ignite_test_scenarios-ignite_client-1  | 8
ignite_test_scenarios-ignite_client-1  | 9
```

Figura 67: Stato della cache a seguito della chiusura di alcuni nodi server, in modalità `IGNORE` - perdita di dati

Al contrario, con le modalità `READ_ONLY_SAFE` e `READ_WRITE_SAFE`, nel caso di lettura o scrittura di un record di una partizione andata persa viene lanciata un'eccezione, come si può vedere in Figura 68 della pagina seguente. Più precisamente, queste due modalità si

differenziano nel caso della scrittura: mentre `READ_WRITE_SAFE` lancia l'eccezione solo nel caso si vada a leggere un record mancante (Figura 69), per `READ_ONLY_SAFE` l'eccezione viene lanciata sempre, anche quando si legge un dato che non è mancante (Figura 70).

```
ignite_test_scenarios-ignite_client-1  | After node close.
ignite_test_scenarios-ignite_client-1  | 0
ignite_test_scenarios-ignite_client-1  | 1
ignite_test_scenarios-ignite_client-1  | 2
ignite_test_scenarios-ignite_client-1  | class org.apache.ignite.internal.processors.cache.CacheInvalidStateException: Failed to execute the cache operation (all partition owners have left the grid, par
tition data has been lost) [cacheName=myCache, partition=3, key=UserKeyCacheObjectImpl [part=3, val=3, hasValBytes=true]]
ignite_test_scenarios-ignite_client-1  | class org.apache.ignite.internal.processors.cache.CacheInvalidStateException: Failed to execute the cache operation (all partition owners have left the grid, par
tition data has been lost) [cacheName=myCache, partition=4, key=UserKeyCacheObjectImpl [part=4, val=4, hasValBytes=false]]
ignite_test_scenarios-ignite_client-1  | 5
ignite_test_scenarios-ignite_client-1  | 6
ignite_test_scenarios-ignite_client-1  | 7
ignite_test_scenarios-ignite_client-1  | 8
ignite_test_scenarios-ignite_client-1  | 9
```

Figura 68: Stato della cache a seguito della chiusura di alcuni nodi server, in modalità `READ_ONLY_SAFE` o `READ_WRITE_SAFE` - perdita di dati con eccezione

```
ignite_test_scenarios-ignite_client-1  | 0
ignite_test_scenarios-ignite_client-1  | 1
ignite_test_scenarios-ignite_client-1  | class org.apache.ignite.internal.processors.cache.CacheInvalidStateException: Failed to execute the cache operation (all partition owners have left the grid, partition data
 has been lost) [cacheName=myCache, partition=2, key=UserKeyCacheObjectImpl [part=2, val=2, hasValBytes=false]]
ignite_test_scenarios-ignite_client-1  | class org.apache.ignite.internal.processors.cache.CacheInvalidStateException: Failed to execute the cache operation (all partition owners have left the grid, partition data
 has been lost) [cacheName=myCache, partition=3, key=UserKeyCacheObjectImpl [part=3, val=3, hasValBytes=false]]
ignite_test_scenarios-ignite_client-1  | class org.apache.ignite.internal.processors.cache.CacheInvalidStateException: Failed to execute the cache operation (all partition owners have left the grid, partition data
 has been lost) [cacheName=myCache, partition=4, key=UserKeyCacheObjectImpl [part=4, val=4, hasValBytes=false]]
ignite_test_scenarios-ignite_client-1  | class org.apache.ignite.internal.processors.cache.CacheInvalidStateException: Failed to execute the cache operation (all partition owners have left the grid, partition data
 has been lost) [cacheName=myCache, partition=5, key=UserKeyCacheObjectImpl [part=5, val=5, hasValBytes=false]]
ignite_test_scenarios-ignite_client-1  | 6
ignite_test_scenarios-ignite_client-1  | 7
ignite_test_scenarios-ignite_client-1  | class org.apache.ignite.internal.processors.cache.CacheInvalidStateException: Failed to execute the cache operation (all partition owners have left the grid, partition data
 has been lost) [cacheName=myCache, partition=8, key=UserKeyCacheObjectImpl [part=8, val=8, hasValBytes=false]]
ignite_test_scenarios-ignite_client-1  | 9
ignite_test_scenarios-ignite_client-1  | Insert after potential partition loss:
ignite_test_scenarios-ignite_client-1  | Exception in thread "main" javax.cache.CacheException: class org.apache.ignite.internal.processors.cache.CacheInvalidStateException: Failed to execute the cache operation (
all partition owners have left the grid, partition data has been lost) [cacheName=myCache, partition=2, key=2]
ignite_test_scenarios-ignite_client-1  |        at org.apache.ignite.internal.processors.cache.GridCacheUtils.convertToCacheException(GridCacheUtils.java:1272)
ignite_test_scenarios-ignite_client-1  |        at org.apache.ignite.internal.processors.cache.IgniteCacheProxyImpl.cacheException(IgniteCacheProxyImpl.java:2099)
ignite_test_scenarios-ignite_client-1  |        at org.apache.ignite.internal.processors.cache.IgniteCacheProxyImpl.put(IgniteCacheProxyImpl.java:1335)
ignite_test_scenarios-ignite_client-1  |        at org.apache.ignite.internal.processors.cache.GatewayProtectedCacheProxy.put(GatewayProtectedCacheProxy.java:867)
ignite_test_scenarios-ignite_client-1  |        at it.unibo.ds.ignite.caching.scenario_1_X_2.Client.main(Client.java:88)
ignite_test_scenarios-ignite_client-1  | Caused by: class org.apache.ignite.internal.processors.cache.CacheInvalidStateException: Failed to execute the cache operation (all partition owners have left the grid, par
tition data has been lost) [cacheName=myCache, partition=2, key=2]
ignite_test_scenarios-ignite_client-1  |        at org.apache.ignite.internal.processors.cache.distributed.dht.GridDhtTopologyFutureAdapter.validateKey(GridDhtTopologyFutureAdapter.java:214)
ignite_test_scenarios-ignite_client-1  |        at org.apache.ignite.internal.processors.cache.distributed.dht.GridDhtTopologyFutureAdapter.validateCache(GridDhtTopologyFutureAdapter.java:133)
ignite_test_scenarios-ignite_client-1  |        at org.apache.ignite.internal.processors.cache.distributed.dht.atomic.GridNearAtomicSingleUpdateFuture.mapOnTopology(GridNearAtomicSingleUpdateFuture.java:415)
ignite_test_scenarios-ignite_client-1  |        at org.apache.ignite.internal.processors.cache.distributed.dht.atomic.GridNearAtomicAbstractUpdateFuture.map(GridNearAtomicAbstractUpdateFuture.java:249)
ignite_test_scenarios-ignite_client-1  |        at org.apache.ignite.internal.processors.cache.distributed.dht.atomic.GridDhtAtomicCache.update0(GridDhtAtomicCache.java:1149)
ignite_test_scenarios-ignite_client-1  |        at org.apache.ignite.internal.processors.cache.distributed.dht.atomic.GridDhtAtomicCache.put0(GridDhtAtomicCache.java:617)
ignite_test_scenarios-ignite_client-1  |        at org.apache.ignite.internal.processors.cache.GridCacheAdapter.put(GridCacheAdapter.java:2487)
ignite_test_scenarios-ignite_client-1  |        at org.apache.ignite.internal.processors.cache.GridCacheAdapter.put(GridCacheAdapter.java:2466)
ignite_test_scenarios-ignite_client-1  |        at org.apache.ignite.internal.processors.cache.IgniteCacheProxyImpl.put(IgniteCacheProxyImpl.java:1332)
ignite_test_scenarios-ignite_client-1  |        ... 2 more
```

Figura 69: Stato della cache a seguito della chiusura di alcuni nodi server - tentativo di scrittura in modalità `READ_WRITE_SAFE`

```
ignite_test_scenarios-ignite_client-1  | After node close.
ignite_test_scenarios-ignite_client-1  | 0
ignite_test_scenarios-ignite_client-1  | 1
ignite_test_scenarios-ignite_client-1  | class org.apache.ignite.internal.processors.cache.CacheInvalidStateException: Failed to execute the cache operation (all partition owners have left the grid, partition data
 has been lost) [cacheName=myCache, partition=2, key=UserKeyCacheObjectImpl [part=2, val=2, hasValBytes=false]]
ignite_test_scenarios-ignite_client-1  | class org.apache.ignite.internal.processors.cache.CacheInvalidStateException: Failed to execute the cache operation (all partition owners have left the grid, partition data
 has been lost) [cacheName=myCache, partition=3, key=UserKeyCacheObjectImpl [part=3, val=3, hasValBytes=false]]
ignite_test_scenarios-ignite_client-1  | class org.apache.ignite.internal.processors.cache.CacheInvalidStateException: Failed to execute the cache operation (all partition owners have left the grid, partition data
 has been lost) [cacheName=myCache, partition=4, key=UserKeyCacheObjectImpl [part=4, val=4, hasValBytes=false]]
ignite_test_scenarios-ignite_client-1  | class org.apache.ignite.internal.processors.cache.CacheInvalidStateException: Failed to execute the cache operation (all partition owners have left the grid, partition data
 has been lost) [cacheName=myCache, partition=5, key=UserKeyCacheObjectImpl [part=5, val=5, hasValBytes=false]]
ignite_test_scenarios-ignite_client-1  | 6
ignite_test_scenarios-ignite_client-1  | 7
ignite_test_scenarios-ignite_client-1  | class org.apache.ignite.internal.processors.cache.CacheInvalidStateException: Failed to execute the cache operation (all partition owners have left the grid, partition data
 has been lost) [cacheName=myCache, partition=8, key=UserKeyCacheObjectImpl [part=8, val=8, hasValBytes=false]]
ignite_test_scenarios-ignite_client-1  | 9
ignite_test_scenarios-ignite_client-1  | Insert after potential partition loss:
ignite_test_scenarios-ignite_client-1  | Exception in thread "main" javax.cache.CacheException: class org.apache.ignite.internal.processors.cache.CacheInvalidStateException: Failed to write to cache (cache is move
d to a read-only state): myCache
ignite_test_scenarios-ignite_client-1  |        at org.apache.ignite.internal.processors.cache.GridCacheUtils.convertToCacheException(GridCacheUtils.java:1272)
ignite_test_scenarios-ignite_client-1  |        at org.apache.ignite.internal.processors.cache.IgniteCacheProxyImpl.cacheException(IgniteCacheProxyImpl.java:2099)
ignite_test_scenarios-ignite_client-1  |        at org.apache.ignite.internal.processors.cache.IgniteCacheProxyImpl.put(IgniteCacheProxyImpl.java:1335)
ignite_test_scenarios-ignite_client-1  |        at org.apache.ignite.internal.processors.cache.GatewayProtectedCacheProxy.put(GatewayProtectedCacheProxy.java:867)
ignite_test_scenarios-ignite_client-1  |        at it.unibo.ds.ignite.caching.scenario_1_X_2.Client.main(Client.java:88)
ignite_test_scenarios-ignite_client-1  | Caused by: class org.apache.ignite.internal.processors.cache.CacheInvalidStateException: Failed to write to cache (cache is moved to a read-only state): myCache
ignite_test_scenarios-ignite_client-1  |        at org.apache.ignite.internal.processors.cache.distributed.dht.GridDhtTopologyFutureAdapter.validateCache(GridDhtTopologyFutureAdapter.java:125)
ignite_test_scenarios-ignite_client-1  |        at org.apache.ignite.internal.processors.cache.distributed.dht.atomic.GridNearAtomicSingleUpdateFuture.mapOnTopology(GridNearAtomicSingleUpdateFuture.java:415)
ignite_test_scenarios-ignite_client-1  |        at org.apache.ignite.internal.processors.cache.distributed.dht.atomic.GridNearAtomicAbstractUpdateFuture.map(GridNearAtomicAbstractUpdateFuture.java:249)
ignite_test_scenarios-ignite_client-1  |        at org.apache.ignite.internal.processors.cache.distributed.dht.atomic.GridDhtAtomicCache.update0(GridDhtAtomicCache.java:1149)
ignite_test_scenarios-ignite_client-1  |        at org.apache.ignite.internal.processors.cache.distributed.dht.atomic.GridDhtAtomicCache.put0(GridDhtAtomicCache.java:617)
ignite_test_scenarios-ignite_client-1  |        at org.apache.ignite.internal.processors.cache.GridCacheAdapter.put(GridCacheAdapter.java:2487)
ignite_test_scenarios-ignite_client-1  |        at org.apache.ignite.internal.processors.cache.GridCacheAdapter.put(GridCacheAdapter.java:2466)
ignite_test_scenarios-ignite_client-1  |        at org.apache.ignite.internal.processors.cache.IgniteCacheProxyImpl.put(IgniteCacheProxyImpl.java:1332)
ignite_test_scenarios-ignite_client-1  |        ... 2 more
```

Figura 70: Stato della cache a seguito della chiusura di alcuni nodi server - tentativo di scrittura in modalità `READ_ONLY_SAFE`

Per l'esecuzione di questo test i nodi server vengono lanciati programmaticamente nello stesso container del nodo client, per questo motivo nel file `.env` il valore del numero di server è impostato a 0.

8.2.7 Native persistence (scenario 2.1)

Il presente test sfrutta Ignite nella sua modalità native persistence, la quale consente di fare affidamento anche sul disco, oltre che alla cache in-memory, per una maggiore affidabilità e prevenzione in caso di failure del sistema. In questo modo, il rischio di perdita di dati si riduce ulteriormente, dal momento che ciascun nodo memorizza una copia di ogni partizione di cui è resposabile in un corrispondente file su disco, rispettando lo stesso formato e la stessa struttura dei dati nella cache in-memory.

In Figura 71 sono mostrati i diversi file relativi alla persistenza che vengono creati con l'esecuzione di questo test. Com'è possibile riscontrare, la struttura è aderente con quella illustrata nel Capitolo 2.2.3. Al percorso specificato nella variabile d'ambiente `IGNITE_HOME`, in questo caso `/opt/ignite` del container, è quindi presente la sotto-directory `work/db`, che tra le altre cose contiene una cartella per ciascun nodo server della topologia. All'interno di essa, vengono memorizzati i file corrispondenti a ciascuna partizione (di default 1024) e contenenti le relative entry. Considerando che i nodi server sono in esecuzione su container separati, il file system di ognuno di essi mantiene solamente i file del nodo specifico.

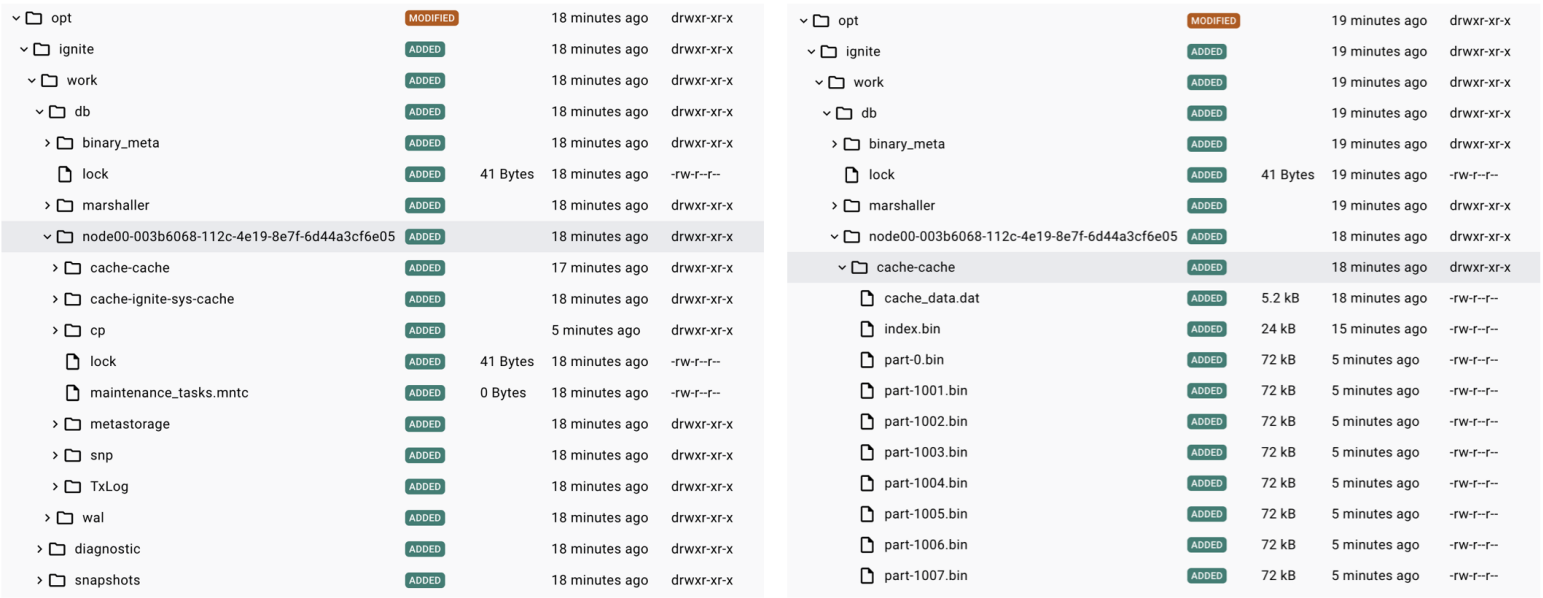

Figura 71: Struttura complessiva delle directory generate in modalità native persistence (a sinistra) e dettaglio dei file di partizione della cache presso uno specifico nodo (a destra)

Entrando nel dettaglio del test, come prima cosa viene popolata una cache con 300 mila entry per osservare la differenza di comportamento a cui si assiste nelle due principali modalità di WAL che si possono scegliere e che differiscono per le garanzie di consistenza che offrono in caso di failure dei nodi. Come facilmente immaginabile ed effettivamente riscontrato, la modalità `FSYNC`, che è quella con garanzie maggiori, impiega circa il doppio rispetto alla modalità `BACKGROUND` (che in questo caso coincide con la modalità `LOG_ONLY` attiva di default), poiché per ogni entry esegue una scrittura sincrona sul disco.

Dopodiché, si procede andando a stoppare alcuni nodi server per poi farli ripartire, per osservare quello che è il principale vantaggio offerto dalla native persistence, vale a dire la capacità dei nodi di riconnettersi al cluster e di recuperare tutti i dati che memorizzavano in cache. Per far ciò, si utilizza un apposito meccanismo di assegnazione del cosiddetto *consistent ID* che assegna al nodo un identificatore univoco globale che sopravvive al riavvio dello stesso. In questo caso, l'effettivo recupero dei dati è da ricondursi unicamente alla feature di native persistence, dal momento che si è provveduto a disabilitare altri meccanismi che avrebbero potutuo raggiungere lo stesso obiettivo senza sfruttare la persistenza, come ad esempio il data rebalancing tra i nodi.

In Figura 72 è mostrato parte dell'output del test, eseguito sulla Macchina 1 con 3 nodi server. Si fa notare che in questo caso, per esigenze implementative, un server viene lanciato come al solito in un container a parte, mentre i restanti due vengono lanciati, fermati e successivamente fatti ripartire nello stesso container del nodo client.

Figura 72: Output dello scenario di test 2.1 - modalità `FSYNC` vs modalità `BACKGROUND`

Ci sono situazioni in cui può essere ragionevole disabilitare momentaneamente il meccanismo di WAL per incrementare le performance, ad esempio negli scenari caratterizzati da un alto volume di scritture in fase di caricamento iniziale dei dati in cache. In questo caso, come si può vedere dall'output riportato in Figura 73, le tempistiche di scrittura sono allineate con quelle della modalità `BACKGROUND` anche in modalità `FSYNC`, a costo di perdere la massima garanzia di consistenza durante questa fase.

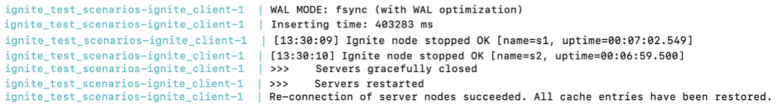

Figura 73: Output dello scenario di test 2.1 - modalità `FSYNC` con disattivazione del WAL in fase di scrittura iniziale

8.2.8 Native persistence con crash dei nodi (scenario 2.2)

Con questo test si vuole indagare più a fondo sulle diverse modalità di WAL impostabili, osservandone gli effettivi livelli di garanzia di consistenza in uno scenario in cui alcuni nodi server falliscono. Dopo l'inserimento di alcune entry nella cache, viene quindi simulato il crash dei 2 nodi server precedentemente avviati. Con il WAL in modalità `FSYNC` o in modalità `BACKGROUND`, a seguito del crash dei server è possibile andare a leggere il contenuto della cache e trovarvi tutti i dati aggiornati. La prima modalità (Figura 74), infatti, offre la massima garanzia di consistenza andando a salvare i cambiamenti anche su disco dopo ogni operazione di scrittura, cosicché i dati non vengano persi nemmeno in caso di crash a livello di sistema operativo. La seconda modalità (Figura 75), invece, salva ogni cambiamento in un cache buffer del sistema operativo o in un memory-mapped file, quindi anche in questo caso non si verifica nessuna perdita di dati in caso di crash dei nodi.

```
ignite_test_scenarios-ignite_server-1  | MODE: fsync
ignite_test_scenarios-ignite_server-2  | MODE: fsync
ignite_test_scenarios-ignite_client-1  | Inserting time: 48896 ms
ignite_test_scenarios-ignite_client-1  | >>>    Server nodes (simulated) crash is going to happen...
ignite_test_scenarios-ignite_client-1  | Disk copies of all cache entries are up-to-date. No data loss occurred following the simulated crash of the nodes.
```

Figura 74: Output dello scenario di test 2.2 - modalità `FSYNC` con crash dei server

```
ignite_test_scenarios-ignite_server-1  | MODE: background
ignite_test_scenarios-ignite_server-2  | MODE: background
ignite_test_scenarios-ignite_client-1  | Inserting time: 31256 ms
ignite_test_scenarios-ignite_client-1  | >>>    Server nodes (simulated) crash is going to happen...
ignite_test_scenarios-ignite_client-1  | Disk copies of all cache entries are up-to-date. No data loss occurred following the simulated crash of the nodes.
```

Figura 75: Output dello scenario di test 2.2 - modalità `BACKGROUND` con crash dei server

Il comportamento appena descritto non si verifica più se si va a disabilitare completamente il meccanismo di WAL, impostando la modalità `NONE`. In questo caso, a seguito del crash dei nodi non sarà possibile ritrovare il contenuto della cache, poiché i cambiamenti ad essa non sono mai stati resi persistenti su disco (Figura 76). Con questa configurazione, infatti, tale operazione ha successo solamente quando i nodi server vengono chiusi in maniera corretta, come si può vedere in Figura 77 della pagina seguente.

```
ignite_test_scenarios-ignite_server-1  | MODE: none
ignite_test_scenarios-ignite_server-2  | MODE: none
ignite_test_scenarios-ignite_client-1  | Inserting time: 18535 ms
ignite_test_scenarios-ignite_client-1  | >>>    Server nodes (simulated) crash is going to happen...
ignite_test_scenarios-ignite_client-1  | Disk copies of the cache entries have not been set because of the simulated crash of the nodes. Data loss occurred.
```

Figura 76: Output dello scenario di test 2.2 - modalità `NONE` con crash dei server

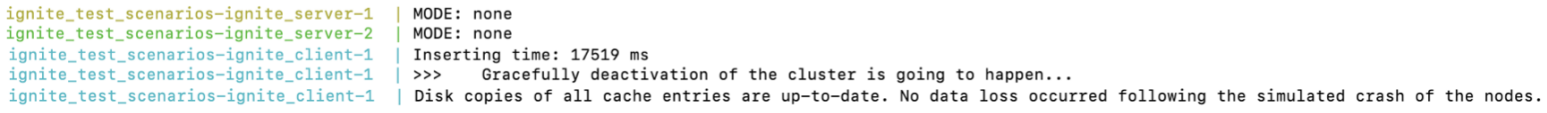

Figura 77: Output dello scenario di test 2.2 - modalità `NONE` con chiusura corretta dei server

Per l'esecuzione del test si è sfruttato il seguente file di configurazione di override, `docker-compose.override.2.2`, per gestire il riavvio dei 2 nodi server e l'avvio del secondo client con cui viene verificata l'eventuale perdita di dati:

```
version: "3.9"

networks:
  cluster_network:
    driver: bridge

services:
  ignite_server:
    environment:
      - TEST_SCENARIO_SERVER=$TEST_SCENARIO_SERVER
      - TEST_SCENARIO_SERVER_PARAMS=$TEST_SCENARIO_SERVER_PARAMS
    command:
      - '/bin/sh'
      - '-c'
      - './gradlew --no-daemon --console=plain $$TEST_SCENARIO_SERVER
          $$TEST_SCENARIO_SERVER_PARAMS ;
         ./gradlew --no-daemon --console=plain $$TEST_SCENARIO_SERVER
            $$TEST_SCENARIO_SERVER_PARAMS'
  ignite_client:
    environment:
      - TEST_SCENARIO_CLIENT=$TEST_SCENARIO_CLIENT
      - TEST_SCENARIO_CLIENT_PARAMS=$TEST_SCENARIO_CLIENT_PARAMS
      - TEST_SCENARIO_CLIENT_2=$TEST_SCENARIO_CLIENT_2
      - TEST_SCENARIO_CLIENT_2_PARAMS=$TEST_SCENARIO_CLIENT_2_PARAMS
    command:
      - '/bin/sh'
      - '-c'
      - './gradlew --no-daemon --console=plain $$TEST_SCENARIO_CLIENT
          $$TEST_SCENARIO_CLIENT_PARAMS ;
         ./gradlew --no-daemon --console=plain $$TEST_SCENARIO_CLIENT_2
            $$TEST_SCENARIO_CLIENT_2_PARAMS'
    depends_on:
      - "ignite_server"
```

8.2.9 SQL API + Cassandra external storage (scenario 3.1)

L'obiettivo di questo test è da una parte quello di utilizzare Ignite nella sua modalità external storage appoggiandosi a un database Cassandra, e dall'altra quello di manipolare i dati qui memorizzati attraverso l'API SQL, grazie alla quale si è in grado di fornire un supporto completo a SQL, non altrimenti possibile in Cassandra e in altri database non relazionali. Cassandra, infatti, in qualità di database NoSQL, utilizza come modello e linguaggio per le interrogazioni CQL, un sottoinsieme di SQL che tra le altre cose non consente di eseguire i join.

In questo scenario, quindi, oltre ai soliti nodi server e client è coinvolto anche il database esterno. Per facilitare l'esecuzione del test è stato predisposto un apposito file di configurazione di override, `docker-compose.override.3.1.yaml`, che va ad estendere la configurazione di base utilizzata per tutti gli altri test introducendo il container

responsabile dell'esecuzione dell'istanza di Cassandra. Tale file si presenta nel seguente modo:

```
version: "3.9"

services:
  cassandra:
    image: bitnami/cassandra:4.0.6
    volumes:
      - ./src/main/resources/data.cql:/docker-entrypoint-initdb.d/data.cql
    networks:
      - cluster_network
  ignite_server:
    environment:
      - TEST_SCENARIO_SERVER=$TEST_SCENARIO_SERVER
    command:
      - '/bin/sh'
      - '-c'
      - '/bin/sleep 60 && ./gradlew --no-daemon --console=plain
          $$TEST_SCENARIO_SERVER'
    depends_on:
      - "cassandra"
  ignite_client:
    environment:
      - TEST_SCENARIO_CLIENT=$TEST_SCENARIO_CLIENT
    command:
      - '/bin/sh'
      - '-c'
      - '/bin/sleep 60 && ./gradlew --no-daemon --console=plain
          $$TEST_SCENARIO_CLIENT'
    depends_on:
      - "cassandra"
```

Il servizio relativo a Cassandra si occupa di avviare l'istanza del DB esterno dall'immagine `bitnami/cassandra`, andando a specificare un volume per il caricamento dei dati che andranno a costituire il dominio di riferimento. Il file `data.cql`, che contiene la definizione dei dati in questione, viene montato nella cartella `docker-entrypoint-initdb.d` del container, che viene letta in automatico all'avvio del servizio per procedere con il caricamento di tutte le tabelle e i dati. Come si può vedere, sono anche stati estesi i servizi relativi ai nodi client e server, in modo da esplicitare la loro dipendenza dal container di Cassandra. Tuttavia, questa specifica non è sufficiente: essa, infatti, aspetta semplicemente che il particolare servizio sia partito, senza premurarsi che sia effettivamente "pronto". Nel caso specifico, tale situazione è problematica dal momento che Cassandra impiega diversi secondi per caricare e istanziare i dati passati in input, pertanto i server e i client non troverebbero ancora i dati da memorizzare/interrogare. Per far fronte a ciò, è stata aggiunta una sleep (tramite l'istruzione `command`) nei container dei server e del client per assicurarsi che quando essi partono, i dati siano già stati caricati correttamente sul DB.

Il dominio scelto modella un ipotetico sistema di prenotazione online di biglietti di concerti. Esso si compone della tabella `USER` contenente i dati anagrafici degli utenti iscritti al sistema, della tabella `CONCERT` dei concerti disponibili e delle relative informazioni, della tabella `TICKET` relativa ai biglietti venduti e della tabella di servizio `TICKET_TYPE` che riassume le tipolgie di biglietto che possono essere vendute.

La Figura 78 mostra l'output del test, eseguito sulla Macchina 1 con 4 nodi server. Oltre alle semplici query esplorative per stamapre il contenuto delle varie tabelle, è stata definita una query di join che mostra il numero di biglietti venduti per ciascuna persona, in ordine decresente. Dal momento che Ignite tratta ogni query di join come se fosse colocated, qualora i dati coinvolti non siano effettivamente co-locati, come in questo caso, è necessario specificare `SqlFieldsQuery.setDistributedJoins(true)`, in caso contrario, come si può riscontrare, è probabile che il risultato restituito dalla query non sia corretto, ovvero sia incompleto.

```
ignite_test_scenarios-ignite_client-1  | +++ USER ENTRIES +++
ignite_test_scenarios-ignite_client-1  |       (userId, firstName, lastName, birthDate, gender)
ignite_test_scenarios-ignite_client-1  | >>>    [2, Franco, Neri, 1991-01-01 00:00:00.0, M]
ignite_test_scenarios-ignite_client-1  | >>>    [1, Mario, Rossi, 1990-01-01 00:00:00.0, M]
ignite_test_scenarios-ignite_client-1  | >>>    [3, Giorgio, Bianchi, 1993-01-01 00:00:00.0, M]
ignite_test_scenarios-ignite_client-1  | >>>    [4, Maria, Rosetti, 1980-01-01 00:00:00.0, F]
ignite_test_scenarios-ignite_client-1  |
ignite_test_scenarios-ignite_client-1  | +++ CONCERT ENTRIES +++
ignite_test_scenarios-ignite_client-1  |       (concertId, name, concertDate, availableTickets)
ignite_test_scenarios-ignite_client-1  | >>>    [2, MetalMilano, 2023-11-01 00:00:00.0, 1000]
ignite_test_scenarios-ignite_client-1  | >>>    [1, RockRoma, 2022-12-12 00:00:00.0, 4000]
ignite_test_scenarios-ignite_client-1  | >>>    [3, PopPadova, 2024-02-20 00:00:00.0, 2000]
ignite_test_scenarios-ignite_client-1  | >>>    [4, ClassicCesena, 2023-01-03 00:00:00.0, 8000]
ignite_test_scenarios-ignite_client-1  |
ignite_test_scenarios-ignite_client-1  | +++ TICKET ENTRIES +++
ignite_test_scenarios-ignite_client-1  |       (ticketId, type, userId, concertId)
ignite_test_scenarios-ignite_client-1  | >>>    [2, super, 2, 1]
ignite_test_scenarios-ignite_client-1  | >>>    [5, base, 4, 2]
ignite_test_scenarios-ignite_client-1  | >>>    [6, base, 4, 2]
ignite_test_scenarios-ignite_client-1  | >>>    [8, vip, 3, 2]
ignite_test_scenarios-ignite_client-1  | >>>    [12, vip, 3, 3]
ignite_test_scenarios-ignite_client-1  | >>>    [16, vip, 3, 4]
ignite_test_scenarios-ignite_client-1  | >>>    [1, base, 1, 1]
ignite_test_scenarios-ignite_client-1  | >>>    [3, base, 1, 1]
ignite_test_scenarios-ignite_client-1  | >>>    [4, vip, 3, 1]
ignite_test_scenarios-ignite_client-1  | >>>    [7, base, 4, 2]
ignite_test_scenarios-ignite_client-1  | >>>    [9, base, 1, 3]
ignite_test_scenarios-ignite_client-1  | >>>    [10, super, 2, 3]
ignite_test_scenarios-ignite_client-1  | >>>    [11, base, 1, 3]
ignite_test_scenarios-ignite_client-1  | >>>    [13, base, 4, 4]
ignite_test_scenarios-ignite_client-1  | >>>    [14, base, 4, 4]
ignite_test_scenarios-ignite_client-1  | >>>    [15, base, 4, 4]
ignite_test_scenarios-ignite_client-1  |
ignite_test_scenarios-ignite_client-1  | +++ TICKET_TYPE ENTRIES +++
ignite_test_scenarios-ignite_client-1  |       (type, price, zone)
ignite_test_scenarios-ignite_client-1  | >>>    [base, 30.99, back]
ignite_test_scenarios-ignite_client-1  | >>>    [super, 45.5, center]
ignite_test_scenarios-ignite_client-1  | >>>    [vip, 79.99, front]
ignite_test_scenarios-ignite_client-1  |
ignite_test_scenarios-ignite_client-1  | +++ NUMBER OF TICKETS SOLD TO EACH PERSON +++
ignite_test_scenarios-ignite_client-1  | [with non-colocated mode enabled - proper way]
ignite_test_scenarios-ignite_client-1  |       (firstName, lastName, numTicketsPurchased)
ignite_test_scenarios-ignite_client-1  | >>>    [Maria, Rosetti, 6]
ignite_test_scenarios-ignite_client-1  | >>>    [Mario, Rossi, 4]
ignite_test_scenarios-ignite_client-1  | >>>    [Giorgio, Bianchi, 4]
ignite_test_scenarios-ignite_client-1  | >>>    [Franco, Neri, 2]
ignite_test_scenarios-ignite_client-1  |
ignite_test_scenarios-ignite_client-1  | [with non-colocated mode disabled - wrong way, possible data loss!]
ignite_test_scenarios-ignite_client-1  |       (firstName, lastName, numTicketsPurchased)
ignite_test_scenarios-ignite_client-1  | >>>    [Mario, Rossi, 4]
ignite_test_scenarios-ignite_client-1  | >>>    [Maria, Rosetti, 4]
ignite_test_scenarios-ignite_client-1  | >>>    [Franco, Neri, 1]
ignite_test_scenarios-ignite_client-1  | >>>    [Giorgio, Bianchi, 1]
```

Figura 78: Output dello scenario di test 3.1

8.2.10 Join distribuiti colocated vs. non-colocated (scenario 3.2)

Lo scopo di questo test è quello di analizzare più a fondo il comportamento dei join distribuiti a seconda della modalità scelta, soffermandosi sulle differenze di performance. Nello specifico, viene eseguita una query di join per stampare il numero di biglietti venduti per ciascun concerto, utilizzando la Macchina 1 e 3 nodi server (Figura 79).

```
ignite_test_scenarios-ignite_client-1  | CONCERT CACHE SIZE: 10 entries
ignite_test_scenarios-ignite_client-1  | TICKET CACHE SIZE: 30 entries
ignite_test_scenarios-ignite_client-1  |
ignite_test_scenarios-ignite_client-1  | +++ NUMBER OF TICKETS SOLD FOR EACH CONCERT [NON-COLOCATED JOIN QUERY] +++
ignite_test_scenarios-ignite_client-1  |       (name, numTicketsSold)
ignite_test_scenarios-ignite_client-1  | >>>    [Concert1, 2]
ignite_test_scenarios-ignite_client-1  | >>>    [Concert2, 1]
ignite_test_scenarios-ignite_client-1  | >>>    [Concert3, 5]
ignite_test_scenarios-ignite_client-1  | >>>    [Concert4, 1]
ignite_test_scenarios-ignite_client-1  | >>>    [Concert6, 5]
ignite_test_scenarios-ignite_client-1  | >>>    [Concert7, 4]
ignite_test_scenarios-ignite_client-1  | >>>    [Concert8, 2]
ignite_test_scenarios-ignite_client-1  | >>>    [Concert9, 7]
ignite_test_scenarios-ignite_client-1  | >>>    [Concert0, 3]
ignite_test_scenarios-ignite_client-1  |
ignite_test_scenarios-ignite_client-1  | Execution time: 2314
```

Figura 79: Output dello scenario di test 3.2

A seconda dei parametri che si passano in input, è possibile eseguire tale scenario sia in modalità non-colocated che in modalità colocated. In quest'ultimo caso, le due tabelle coinvolte nella co-locazione sono `CONCERT` e `TICKET`, così facendo i biglietti venduti per un dato concerto vengono memorizzati nello stesso nodo in cui si trova il concerto stesso. Per osservare effettivamente questo comportamento, viene eseguita un'apposita scan query locale su ciascun nodo server, in modo che ognuno di essi stampi unicamente le entry che memorizza. Come si può vedere dalla Figura 80, in modalità colocated ciascun nodo mantiene esattamente i biglietti relativi ai soli concerti lì presenti, a differenza della modalità non-colocated.

```
ignite_test_scenarios-ignite_server-1 | +++ COLOCATED LOCAL SCAN QUERIES +++
ignite_test_scenarios-ignite_server-1 | CONCERT CACHE: 3 entries
ignite_test_scenarios-ignite_server-1 |       (concertId, name, concertDate, availableTickets)
ignite_test_scenarios-ignite_server-1 | >>>    [3, Concert3, Sat May 16 03:16:34 GMT 2020, 7031]
ignite_test_scenarios-ignite_server-1 | >>>    [5, Concert5, Sat Dec 29 22:33:26 GMT 2018, 9478]
ignite_test_scenarios-ignite_server-1 | >>>    [7, Concert7, Wed Sep 28 16:59:02 GMT 2022, 917]
ignite_test_scenarios-ignite_server-1 |
ignite_test_scenarios-ignite_server-1 | TICKET CACHE: 9 entries
ignite_test_scenarios-ignite_server-1 |       (ticketId, type, userId, concertId)
ignite_test_scenarios-ignite_server-1 | >>>    [21, base, 7, 3]
ignite_test_scenarios-ignite_server-1 | >>>    [9, base, 7, 3]
ignite_test_scenarios-ignite_server-1 | >>>    [20, super, 4, 3]
ignite_test_scenarios-ignite_server-1 | >>>    [5, base, 2, 3]
ignite_test_scenarios-ignite_server-1 | >>>    [19, vip, 5, 3]
ignite_test_scenarios-ignite_server-1 | >>>    [2, vip, 8, 7]
ignite_test_scenarios-ignite_server-1 | >>>    [4, base, 5, 7]
ignite_test_scenarios-ignite_server-1 | >>>    [8, base, 4, 7]
ignite_test_scenarios-ignite_server-1 | >>>    [7, vip, 3, 7]
ignite_test_scenarios-ignite_server-1 |
ignite_test_scenarios-ignite_server-2 | +++ COLOCATED LOCAL SCAN QUERIES +++
ignite_test_scenarios-ignite_server-2 | CONCERT CACHE: 1 entries
ignite_test_scenarios-ignite_server-2 |       (concertId, name, concertDate, availableTickets)
ignite_test_scenarios-ignite_server-2 | >>>    [6, Concert6, Tue Sep 29 19:36:59 GMT 2020, 3158]
ignite_test_scenarios-ignite_server-2 |
ignite_test_scenarios-ignite_server-2 | TICKET CACHE: 5 entries
ignite_test_scenarios-ignite_server-2 |       (ticketId, type, userId, concertId)
ignite_test_scenarios-ignite_server-2 | >>>    [23, vip, 7, 6]
ignite_test_scenarios-ignite_server-2 | >>>    [18, vip, 3, 6]
ignite_test_scenarios-ignite_server-2 | >>>    [25, super, 9, 6]
ignite_test_scenarios-ignite_server-2 | >>>    [3, super, 0, 6]
ignite_test_scenarios-ignite_server-2 | >>>    [0, base, 3, 6]
ignite_test_scenarios-ignite_server-2 |
ignite_test_scenarios-ignite_server-3 | +++ COLOCATED LOCAL SCAN QUERIES +++
ignite_test_scenarios-ignite_server-3 | CONCERT CACHE: 6 entries
ignite_test_scenarios-ignite_server-3 |       (concertId, name, concertDate, availableTickets)
ignite_test_scenarios-ignite_server-3 | >>>    [0, Concert0, Mon Sep 26 04:20:25 GMT 2016, 6496]
ignite_test_scenarios-ignite_server-3 | >>>    [1, Concert1, Mon Sep 30 04:53:00 GMT 2019, 4006]
ignite_test_scenarios-ignite_server-3 | >>>    [2, Concert2, Sun Jul 03 05:10:09 GMT 2022, 9422]
ignite_test_scenarios-ignite_server-3 | >>>    [4, Concert4, Tue Sep 18 11:47:53 GMT 2018, 5650]
ignite_test_scenarios-ignite_server-3 | >>>    [8, Concert8, Sat Aug 21 14:57:02 GMT 2021, 786]
ignite_test_scenarios-ignite_server-3 | >>>    [9, Concert9, Sun Jun 06 02:47:06 GMT 2021, 4594]
ignite_test_scenarios-ignite_server-3 |
ignite_test_scenarios-ignite_server-3 | TICKET CACHE: 16 entries
ignite_test_scenarios-ignite_server-3 |       (ticketId, type, userId, concertId)
ignite_test_scenarios-ignite_server-3 | >>>    [26, base, 8, 0]
ignite_test_scenarios-ignite_server-3 | >>>    [16, super, 9, 0]
ignite_test_scenarios-ignite_server-3 | >>>    [11, vip, 0, 0]
ignite_test_scenarios-ignite_server-3 | >>>    [1, super, 3, 1]
ignite_test_scenarios-ignite_server-3 | >>>    [12, vip, 9, 1]
ignite_test_scenarios-ignite_server-3 | >>>    [29, base, 5, 2]
ignite_test_scenarios-ignite_server-3 | >>>    [14, base, 1, 4]
ignite_test_scenarios-ignite_server-3 | >>>    [22, super, 4, 8]
ignite_test_scenarios-ignite_server-3 | >>>    [10, vip, 0, 8]
ignite_test_scenarios-ignite_server-3 | >>>    [28, base, 1, 9]
ignite_test_scenarios-ignite_server-3 | >>>    [6, vip, 4, 9]
ignite_test_scenarios-ignite_server-3 | >>>    [13, super, 7, 9]
ignite_test_scenarios-ignite_server-3 | >>>    [27, vip, 2, 9]
ignite_test_scenarios-ignite_server-3 | >>>    [15, base, 8, 9]
ignite_test_scenarios-ignite_server-3 | >>>    [17, base, 3, 9]
ignite_test_scenarios-ignite_server-3 | >>>    [24, super, 5, 9]
ignite_test_scenarios-ignite_server-3 |
```

```
ignite_test_scenarios-ignite_server-1 | +++ NON-COLOCATED LOCAL SCAN QUERIES +++
ignite_test_scenarios-ignite_server-1 | CONCERT CACHE: 1 entries
ignite_test_scenarios-ignite_server-1 |       (concertId, name, concertDate, availableTickets)
ignite_test_scenarios-ignite_server-1 | >>>    [3, Concert3, Sat May 16 03:16:34 GMT 2020, 7031]
ignite_test_scenarios-ignite_server-1 |
ignite_test_scenarios-ignite_server-1 | TICKET CACHE: 5 entries
ignite_test_scenarios-ignite_server-1 |       (ticketId, type, userId, concertId)
ignite_test_scenarios-ignite_server-1 | >>>    [3, super, 0, 6]
ignite_test_scenarios-ignite_server-1 | >>>    [13, super, 7, 9]
ignite_test_scenarios-ignite_server-1 | >>>    [17, base, 3, 9]
ignite_test_scenarios-ignite_server-1 | >>>    [25, super, 9, 6]
ignite_test_scenarios-ignite_server-1 | >>>    [26, base, 8, 0]
ignite_test_scenarios-ignite_server-1 |
ignite_test_scenarios-ignite_server-2 | +++ NON-COLOCATED LOCAL SCAN QUERIES +++
ignite_test_scenarios-ignite_server-2 | CONCERT CACHE: 3 entries
ignite_test_scenarios-ignite_server-2 |       (concertId, name, concertDate, availableTickets)
ignite_test_scenarios-ignite_server-2 | >>>    [2, Concert2, Sun Jul 03 05:10:09 GMT 2022, 9422]
ignite_test_scenarios-ignite_server-2 | >>>    [4, Concert4, Tue Sep 18 11:47:53 GMT 2018, 5650]
ignite_test_scenarios-ignite_server-2 | >>>    [8, Concert8, Sat Aug 21 14:57:02 GMT 2021, 786]
ignite_test_scenarios-ignite_server-2 |
ignite_test_scenarios-ignite_server-2 | TICKET CACHE: 11 entries
ignite_test_scenarios-ignite_server-2 |       (ticketId, type, userId, concertId)
ignite_test_scenarios-ignite_server-2 | >>>    [2, vip, 8, 7]
ignite_test_scenarios-ignite_server-2 | >>>    [4, base, 5, 7]
ignite_test_scenarios-ignite_server-2 | >>>    [8, base, 4, 7]
ignite_test_scenarios-ignite_server-2 | >>>    [10, vip, 0, 8]
ignite_test_scenarios-ignite_server-2 | >>>    [11, vip, 0, 0]
ignite_test_scenarios-ignite_server-2 | >>>    [12, vip, 9, 1]
ignite_test_scenarios-ignite_server-2 | >>>    [15, base, 8, 9]
ignite_test_scenarios-ignite_server-2 | >>>    [18, vip, 3, 6]
ignite_test_scenarios-ignite_server-2 | >>>    [19, vip, 5, 3]
ignite_test_scenarios-ignite_server-2 | >>>    [21, base, 7, 3]
ignite_test_scenarios-ignite_server-2 | >>>    [23, vip, 7, 6]
ignite_test_scenarios-ignite_server-2 |
ignite_test_scenarios-ignite_server-3 | +++ NON-COLOCATED LOCAL SCAN QUERIES +++
ignite_test_scenarios-ignite_server-3 | CONCERT CACHE: 6 entries
ignite_test_scenarios-ignite_server-3 |       (concertId, name, concertDate, availableTickets)
ignite_test_scenarios-ignite_server-3 | >>>    [0, Concert0, Mon Sep 26 04:20:25 GMT 2016, 6496]
ignite_test_scenarios-ignite_server-3 | >>>    [1, Concert1, Mon Sep 30 04:53:00 GMT 2019, 4006]
ignite_test_scenarios-ignite_server-3 | >>>    [5, Concert5, Sat Dec 29 22:33:26 GMT 2018, 9478]
ignite_test_scenarios-ignite_server-3 | >>>    [6, Concert6, Tue Sep 29 19:36:59 GMT 2020, 3158]
ignite_test_scenarios-ignite_server-3 | >>>    [7, Concert7, Wed Sep 28 16:59:02 GMT 2022, 917]
ignite_test_scenarios-ignite_server-3 | >>>    [9, Concert9, Sun Jun 06 02:47:06 GMT 2021, 4594]
ignite_test_scenarios-ignite_server-3 |
ignite_test_scenarios-ignite_server-3 | TICKET CACHE: 14 entries
ignite_test_scenarios-ignite_server-3 |       (ticketId, type, userId, concertId)
ignite_test_scenarios-ignite_server-3 | >>>    [0, base, 3, 6]
ignite_test_scenarios-ignite_server-3 | >>>    [1, super, 3, 1]
ignite_test_scenarios-ignite_server-3 | >>>    [5, base, 2, 3]
ignite_test_scenarios-ignite_server-3 | >>>    [6, vip, 4, 9]
ignite_test_scenarios-ignite_server-3 | >>>    [7, vip, 3, 7]
ignite_test_scenarios-ignite_server-3 | >>>    [9, base, 7, 3]
ignite_test_scenarios-ignite_server-3 | >>>    [14, base, 1, 4]
ignite_test_scenarios-ignite_server-3 | >>>    [16, super, 9, 0]
ignite_test_scenarios-ignite_server-3 | >>>    [20, super, 4, 3]
ignite_test_scenarios-ignite_server-3 | >>>    [22, super, 4, 8]
ignite_test_scenarios-ignite_server-3 | >>>    [24, super, 5, 9]
ignite_test_scenarios-ignite_server-3 | >>>    [27, vip, 2, 9]
ignite_test_scenarios-ignite_server-3 | >>>    [28, base, 1, 9]
ignite_test_scenarios-ignite_server-3 | >>>    [29, base, 5, 2]
ignite_test_scenarios-ignite_server-3 |
```

Figura 80: Output dei nodi server dello scenario di test 3.2 - modalità colocated (a sinistra) vs. non-colocated (a destra)

Per riscontrare al meglio le differenze di performance tra le due modalità, sono state condotte diverse esecuzioni variando sia il numero di nodi server di cui si compone il cluster, sia il numero di entry memorizzate nelle diverse cache, utilizzando la Macchina

1. Dalla Figura 81 si nota come, a prescindere dal numero di nodi e dal numero di entry, la modalità colocated impiega quasi sempre circa la metà della rispettiva modalità non-colocated. All'aumentare del numero di nodi, il tempo cresce per entrambe le modalità e per un qualunque numero di entry, così come cresce all'aumentare del numero di entry, per entrambe le modalità e per un qualsiasi numero di nodi.

# nodes	NON-COLOCATED (50, 5000)	COLOCATED (50, 5000)	NON-COLOCATED (500, 50000)	COLOCATED (500, 50000)	NON-COLOCATED (5000, 500000)	COLOCATED (5000, 500000)
2	2231	1135	2882	1941	9920	7284
4	2237	1319	5603	3120	22177	12440
6	3911	1897	6645	3749	41157	19143

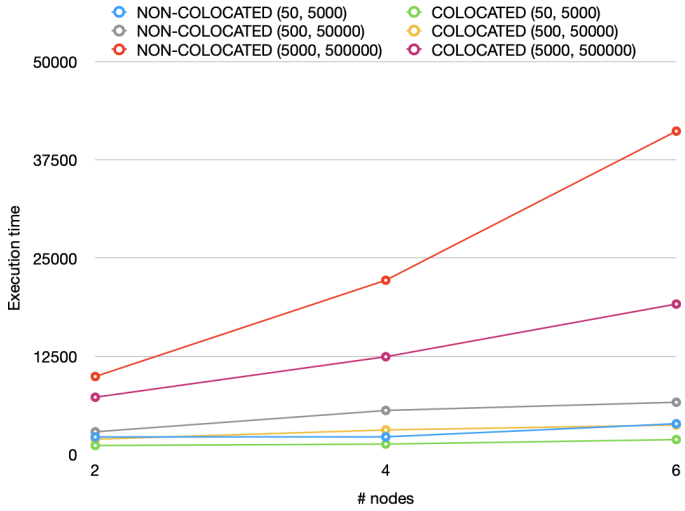

Figura 81: Tabella con i tempi (in millisecondi) registrati nelle due modalità al variare del numero di nodi e del numero di entry (in alto) e grafico con il dettaglio dell'andamento dei tempi (in basso)

8.2.11 Distributed task session (scenario 4.1)

Lo scopo di questo test è quello di sfruttare l'API standard messa a disposizione da Ignite per effettuare computazioni distribuite sui nodi del cluster. Come task si è scelto di calcolare il *tf-idf* (*i.e.* term frequency–inverse document frequency), una funzione spesso usata nell'ambito dell'*information retrivial* per quantificare l'importanza di un certo termine rispetto a una collezione (o *corpus*) di documenti. L'intuizione alla base di tale misurazione sta nel fatto che un termine risulta importante, e quindi discriminante, per un certo documento tante più sono le sue occorrenze in esso, ma allo stesso tempo tale importanza è inversamente proporzionale al numero di volte in cui il termine compare complessivamente nel corpus. Esistono varie formulazioni di questa metrica, quella usata in questo caso è la seguente:

$$(tf\text{-}idf)_{i,j} = tf_{i,j} \times idf_i = \frac{n_{i,j}}{\mid d_j \mid} \times \log_{10} \frac{\mid D \mid}{\mid \{d : i \in d\} \mid} \tag{2}$$

con:

- i: termine
- j: documento
- $n_{i,j}$: numero di occorrenze del termine i nel documento j
- $\mid d_j \mid$: numero complessivo di termini nel documento j
- $\mid D \mid$: numero di documenti nel corpus
- $\mid \{d : i \in d\} \mid$: numero di documenti che contengono il termine i

Come input si è scelto il corpus formato dalle tre Cantiche della *Divina Commedia* di Dante. Per ciascuna di esse viene creato un job separato che viene poi eseguito su un certo nodo remoto mediante il metodo `apply()` dell'interfaccia `IgniteCompute`. Ignite gestisce in maniera automatica il load balancing, pertanto si ha sempre la garanzia che i job creati siano equamente distributi tra i nodi server del cluster.

Considerata la modalità di suddivisione del task scelta (*i.e.* per documento), per portare avanti correttamente la computazione distribuita è necessaria una qualche forma di coordinazione tra i job. A tale scopo si è fatto uso della *distributed task session*, astrazione che permette a job facenti parte dello stesso task di interagire tra di loro e scambiarsi informazioni. In particolare, la coordinazione è stata ottenuta mediante la definizione di specifici attributi, attraverso il metodo `setAttribute()` dell'interfaccia `ComputeTaskSession`, per i quali gli altri job si mettono in attesa tramite il metodo `waitForAttribute()`, realizzando così la sincronizzazione.

In Figura 82 viene mostrato un sample dell'output del task, che consiste nel valore di tf-idf di ogni parola in ciascun documento, eseguito con la Macchina 1 e 3 nodi server.

```
ignite_test_scenarios-ignite_server-3  | Computation is going to be performed on this node (node ID: d5683bb3-9e37-485b-9215-77d54a0cfd99)
ignite_test_scenarios-ignite_server-2  | Computation is going to be performed on this node (node ID: 4850b65d-b2ba-4ff6-98ee-06c8a56b467c)
ignite_test_scenarios-ignite_server-1  | Computation is going to be performed on this node (node ID: 826d871c-9e32-4be4-962b-c4831aba75fe)
ignite_test_scenarios-ignite_client-1  | Sample output:
ignite_test_scenarios-ignite_client-1  |       (doc, word): tf-idf
ignite_test_scenarios-ignite_client-1  | >>>    (inferno, ): 0.0
ignite_test_scenarios-ignite_client-1  | >>>    (inferno, retroso): 1.3989364179899795E-5
ignite_test_scenarios-ignite_client-1  | >>>    (inferno, savvalla): 5.163058085254244E-6
ignite_test_scenarios-ignite_client-1  | >>>    (purgatorio, ): 0.0
ignite_test_scenarios-ignite_client-1  | >>>    (purgatorio, guarde): 1.3972158097682513E-5
ignite_test_scenarios-ignite_client-1  | >>>    (purgatorio, savvalla): 5.156707832250241E-6
ignite_test_scenarios-ignite_client-1  | >>>    (paradiso, ): 0.0
ignite_test_scenarios-ignite_client-1  | >>>    (paradiso, poema): 2.845087982824463E-5
ignite_test_scenarios-ignite_client-1  | >>>    (paradiso, guardi): 0.0
ignite_test_scenarios-ignite_client-1  | Execution time: 5357 ms
```

Figura 82: Output dello scenario di test 4.1

8.2.12 MapReduce API (scenario 4.2)

Con questo test si è voluto implementare il medesimo task dello scenario precedente, sfruttando questa volta il paradigma MapReduce e la relativa API. Dal punto di vista logico, il task può essere decomposto nella sequenza di job seguente:

- **Job 1**: calcola il primo termine di tf-idf, vale a dire la frequenza di ogni parola in ogni documento.
  - **Map**: esegue un pre-processing di base rimuovendo tutti i caratteri di punteggiatura per poi estrapolare ogni parola della riga di input. Per ragioni di performance, include una fase di combine che inizia a calcolare il numero di occorrenze di ciascuna parola nel rispettivo documento, nella partizione corrente.
  - **Reduce**: calcola la frequenza di ogni parola in ogni documento.
- **Job 2**: calcola il numero complessivo di parole di ogni documento.
  - **Map**: riscrive semplicemente l'output del job precedente.
  - **Reduce**: calcola il numero complessivo di parole di ogni documento.
- **Job 3**: calcola in quanti documenti del corpus compare ogni parola.
  - **Map**: inizia il calcolo.
  - **Reduce**: calcola il numero di documenti in cui ogni parola compare.
- **Job 4**: calcola il valore di tf-idf per ogni parola di ogni documento del corpus.
  - **Map**: calcola il tf-idf.
  - **Reduce**: non fa nulla.

Dal punto di vista implementativo, ogni job è stato modellato estendendo dalla classe `ComputeTaskSplitAdapter`, utility che assegna automaticamente i job ai nodi e che tramite il metodo `split()` permette di specificare in che modo partizionare i job di map sulla base dei dati di input. Nella fattispecie, le righe in input ad ogni fase di map dei vari job sopra descritti vengono equamente suddivisi coerentemente alla dimensione del cluster: anche in questo caso interviene il load balancing automatico a distribuire i job di map appena creati in maniera equa sui vari nodi server.

In Figura 83 della pagina seguente viene mostrato un sample dell'output del task, eseguito con la Macchina 1 e 3 nodi server. Tale approccio ha il vantaggio di essere più "general purpose" e flessibile rispetto a quello dello scenario precedente, nel quale la logica di suddivisione della computazione prevedeva la creazione di un job diverso per ogni documento nel corpus, cosa che potrebbe divenire problematica per corpus di elevate dimensioni.

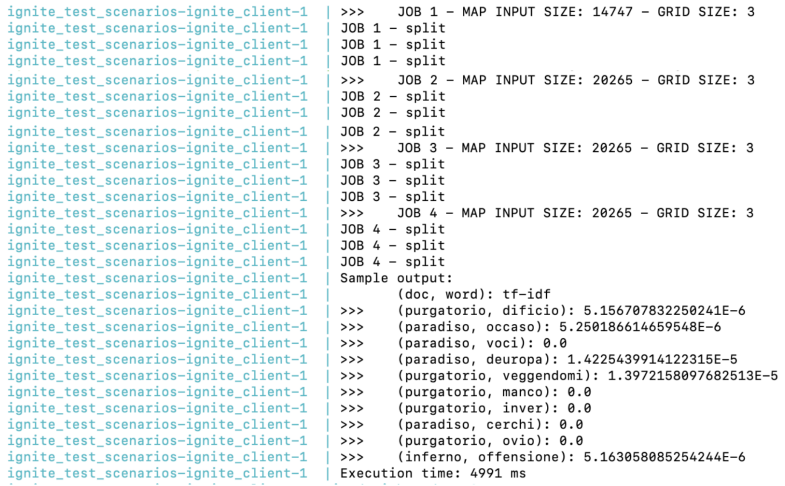

Figura 83: Output dello scenario di test 4.2

8.2.13 Checkpointing (scenario 4.3 + scenario 4.2)

Oltre alle tecniche di sincronizzazione e di coordinazione implementabili tramite la gestione degli attributi, la distributed task session mette a disposizione anche un utile meccanismo di checkpointing tramite il quale è possibile salvare in un qualsiasi istante il progresso della computazione distribuita. Quando avviene una failure di qualsiasi natura, come un'interruzione di rete o un crash vero e proprio di un nodo, Ignite aziona un meccanismo di fault tolerance automatico andando a sottomettere nuovamente il job che era in esecuzione al momento del fault su un altro nodo del cluster (*automatic job failover*). Tale procedura può essere ulteriormente ottimizzata mettendo in campo il checkpointing, per far sì che al momento della riesecuzione il job non venga fatto partire da capo, bensì dal punto esatto in cui era stato interrotto.

Per testare questo comportamento è stato implementato un task che si occupa di calcolare i numeri primi inferiori a un certo numero dato in input. Per enfatizzare la computazione distribuita e renderla più consistente, sebbene esistano algoritmi molto più efficienti in termini di complessità di calcolo, si è scelto di proposito un approccio naïf che va di volta in volta a controllare se ogni numero minore del limite scelto sia primo o meno. Altre ottimizzazioni a livello di logica di computazione sono state volutamente ignorate.

Per simulare la failure, viene lanciata l'eccezione `ComputeJobFailoverException`, automaticamente intercettata e gestita da Ignite nel modo appena descritto. Per quanto riguarda la gestione del checkpoint, invece, ad ogni iterazione viene chiamato il metodo `saveCheckpoint()` dell'interfaccia `ComputeTaskSession` per salvare l'indice relativo all'iterazione successiva da cui dovrà ripartire l'esecuizione in caso di failure. In quest'ultimo caso, l'indice in questione viene recuperato tramite il metodo `loadCheckpoint()`.

In Figura 84 sono mostrati i risultati del test eseguito usando la Macchina 1 con 2 nodi server e impostando come limite 500 mila. Come si può vedere, la versione che utilizza il checkpointing risulta essere più efficiente rispetto all'altra.

```
ignite_test_scenarios-ignite_server-2  | >>>    Job will be failed over to another server node
ignite_test_scenarios-ignite_server-1  | >>>    Computation restarted from checkpoint
ignite_test_scenarios-ignite_client-1  | +++ COMPUTATION WITH CHECKPOINT +++
ignite_test_scenarios-ignite_client-1  | Primes numbers smaller than 500000 are 41538
ignite_test_scenarios-ignite_client-1  | Sample output: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
ignite_test_scenarios-ignite_client-1  | Execution time: 211372 ms
```

```
ignite_test_scenarios-ignite_server-1  | >>>    Job will be failed over to another server node
ignite_test_scenarios-ignite_client-1  | +++ COMPUTATION WITHOUT CHECKPOINT +++
ignite_test_scenarios-ignite_client-1  | Primes numbers smaller than 500000 are 41538
ignite_test_scenarios-ignite_client-1  | Sample output: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
ignite_test_scenarios-ignite_client-1  | Execution time: 344640 ms
```

Figura 84: Output dello scenario di test 4.3 - con checkpointing (a sinistra) vs. senza checkpointing (a destra)

Tuttavia, ci sono casi in cui il meccanismo di checkpointing potrebbe non convenire: per mostrarlo è stato rieseguito il test dello scenario 4.2 includendo una failure in ogni fase di map del Job 4. Come si nota dai risultati in Figura 85, i tempi necessari al salvataggio e al caricamento del risultato parziale nella cache di checkpointing superano di gran lunga quello necessario alla computazione in sé. In questo caso specifico, ciò è dovuto al fatto che il risultato parziale del job è di dimensioni considerevoli, in quanto consiste di una entry per ogni coppia parola-documento di cui si sta calcolando il tf-idf. È pertanto necessario valutare di volta in volta, in base alla natura specifica del task, se sia conveniente o meno prevedere il meccanismo di checkpointing.

```
ignite_test_scenarios-ignite_server-1  | JOB 4 (with simulated failure and checkpointing) - MAP - Checkpoint load time: 131 ms
ignite_test_scenarios-ignite_server-1  | JOB 4 (with simulated failure and checkpointing) - MAP - Compute time: 60 ms
ignite_test_scenarios-ignite_server-1  | JOB 4 (with simulated failure and checkpointing) - MAP - Checkpoint save time: 616 ms
ignite_test_scenarios-ignite_server-1  | >>>    A (simulated) node crash happened. Map job will be failed over to another node.
ignite_test_scenarios-ignite_server-1  | JOB 4 (with simulated failure and checkpointing) - MAP - Checkpoint load time: 197 ms
ignite_test_scenarios-ignite_server-1  | JOB 4 (with simulated failure and checkpointing) - MAP - Execute from checkpoint
ignite_test_scenarios-ignite_server-2  | JOB 4 (with simulated failure and checkpointing) - MAP - Checkpoint load time: 73 ms
ignite_test_scenarios-ignite_server-2  | JOB 4 (with simulated failure and checkpointing) - MAP - Compute time: 14 ms
ignite_test_scenarios-ignite_server-2  | JOB 4 (with simulated failure and checkpointing) - MAP - Checkpoint save time: 250 ms
ignite_test_scenarios-ignite_server-2  | >>>    A (simulated) node crash happened. Map job will be failed over to another node.
ignite_test_scenarios-ignite_server-2  | JOB 4 (with simulated failure and checkpointing) - MAP - Checkpoint load time: 141 ms
ignite_test_scenarios-ignite_server-2  | JOB 4 (with simulated failure and checkpointing) - MAP - Execute from checkpoint
ignite_test_scenarios-ignite_server-3  | JOB 4 (with simulated failure and checkpointing) - MAP - Checkpoint load time: 254 ms
ignite_test_scenarios-ignite_server-3  | JOB 4 (with simulated failure and checkpointing) - MAP - Compute time: 39 ms
ignite_test_scenarios-ignite_server-3  | JOB 4 (with simulated failure and checkpointing) - MAP - Checkpoint save time: 415 ms
ignite_test_scenarios-ignite_server-3  | >>>    A (simulated) node crash happened. Map job will be failed over to another node.
ignite_test_scenarios-ignite_server-3  | JOB 4 (with simulated failure and checkpointing) - MAP - Checkpoint load time: 354 ms
ignite_test_scenarios-ignite_server-3  | JOB 4 (with simulated failure and checkpointing) - MAP - Execute from checkpoint
```

```
ignite_test_scenarios-ignite_client-1  | >>>    JOB 1 - MAP INPUT SIZE: 14747 - GRID SIZE: 3
ignite_test_scenarios-ignite_client-1  | JOB 1 - split
ignite_test_scenarios-ignite_client-1  | JOB 1 - split
ignite_test_scenarios-ignite_client-1  | JOB 1 - split
ignite_test_scenarios-ignite_client-1  | >>>    JOB 2 - MAP INPUT SIZE: 20265 - GRID SIZE: 3
ignite_test_scenarios-ignite_client-1  | JOB 2 - split
ignite_test_scenarios-ignite_client-1  | JOB 2 - split
ignite_test_scenarios-ignite_client-1  | JOB 2 - split
ignite_test_scenarios-ignite_client-1  | >>>    JOB 3 - MAP INPUT SIZE: 20265 - GRID SIZE: 3
ignite_test_scenarios-ignite_client-1  | JOB 3 - split
ignite_test_scenarios-ignite_client-1  | JOB 3 - split
ignite_test_scenarios-ignite_client-1  | JOB 3 - split
ignite_test_scenarios-ignite_client-1  | >>>    JOB 4 - MAP INPUT SIZE: 20265 - GRID SIZE: 3
ignite_test_scenarios-ignite_client-1  | JOB 4 (with simulated failure and checkpointing) - split
ignite_test_scenarios-ignite_client-1  | JOB 4 (with simulated failure and checkpointing) - split
ignite_test_scenarios-ignite_client-1  | JOB 4 (with simulated failure and checkpointing) - split
ignite_test_scenarios-ignite_client-1  | Sample output:
ignite_test_scenarios-ignite_client-1  |       (doc, word): tf-idf
ignite_test_scenarios-ignite_client-1  | >>>    (purgatorio, dificio): 5.156707832250241E-6
ignite_test_scenarios-ignite_client-1  | >>>    (paradiso, occaso): 5.250186614659548E-6
ignite_test_scenarios-ignite_client-1  | >>>    (paradiso, voci): 0.0
ignite_test_scenarios-ignite_client-1  | >>>    (paradiso, deuropa): 1.4225439914122315E-5
ignite_test_scenarios-ignite_client-1  | >>>    (purgatorio, veggendomi): 1.3972158097682513E-5
ignite_test_scenarios-ignite_client-1  | >>>    (purgatorio, manco): 0.0
ignite_test_scenarios-ignite_client-1  | >>>    (purgatorio, inver): 0.0
ignite_test_scenarios-ignite_client-1  | >>>    (paradiso, cerchi): 0.0
ignite_test_scenarios-ignite_client-1  | >>>    (purgatorio, ovio): 0.0
ignite_test_scenarios-ignite_client-1  | >>>    (inferno, offensione): 5.163058085254244E-6
ignite_test_scenarios-ignite_client-1  | Execution time: 16442 ms
```

Figura 85: Output dello scenario di test 4.2 - con failure e checkpointing in Job 4

8.2.14 Task-data colocation (scenario 4.4)

In qualità di data grid distribuito, è possibile sfruttare il cluster Ignite facendo in modo che una certa computazione venga inviata e quindi eseguita direttamente laddove i dati coinvolti risiedono. Questo implica che non vi sia alcuno spostamento di dati, con un conseguente beneficio in termini di performance.

Utilizzando il dominio relativo al sistema di prenotazione di biglietti dei concerti precedentemente definito, è stato delineato un task con lo scopo di calcolare il ricavato

complessivo di ciascun concerto derivante dalla vendita dei biglietti. Per evidenziare l'impatto che l'implementazione basata sulla co-locazione tra task e dati ha sui tempi di esecuzione, è stata confrontata con la rispettiva versione non co-locata. Inoltre, sono state messe a confronto tra di loro diverse strategie di co-locazione. Di seguito si descrivono i vari task definiti allo scopo, modellati tramite classi che implementano l'interfaccia `IgniteCallable` che abilita l'esecuzione distribuita.

- `ConcertIncomeByKeyTask`: è il metodo più diretto che consente di creare un nuovo task per ogni chiave della cache `Concert` e di inviarlo tramite il metodo `affinityCall()` direttamente al nodo che memorizza quella determinata chiave.

- `ConcertIncomeByPartitionNotScalableTask`: si tratta di una versione più ottimizzata che al posto delle singole chiavi permette di specificare direttamente la partizione in cui si trovano le chiavi di interesse. Il metodo `affinityCall()` si occuperà quindi di inviare il task al nodo che contiene quella partizione. Tale approccio è adottabile dal momento che, come si è spiegato nello scenario 3.2, è stata precedentemente effettuata una co-locazione tra le cache `Concert` e `Ticket`, pertanto tutti i biglietti venduti per un certo concerto si trovano sulla stessa partizione del concerto stesso. Per come è stata impostata la logica di questo task, per ottenere un risultato corretto è necessario che non vi sia più di un concerto memorizzato su ciascuna partizione. Ciò è sempre garantito se il numero di concerti si mantiene inferiore o uguale al numero di partizioni, che di default sono 1024. All'occorrenza, è possibile andare a cambiare questo valore agendo sulla proprietà `affinity` della `CacheConfiguration` (specificata nel file di configurazione XML dei nodi server), e specificare così il numero di partizioni che si vuole far usare all'affinity function implementata di default, basata sull'algoritmo di Rendezvous Hashing. Tuttavia, non si tratta di un approccio scalabile all'aumentare del numero di concerti registrati nel sistema nel tempo, inoltre, un tale accrescimento di partizioni non farebbe altro che aumentare il numero di task da creare, con conseguente deterioramento delle performance.

- `ConcertIncomeByPartitionTask`: per risolvere i problemi appena citati, è possibile generalizzare la versione precedente, rivedendo leggermente la logica di computazione in modo che si mantenga corretta anche in presenza di più concerti per ogni partizione. In questo caso non verranno mai creati più di 1024 task, indipendentemente dal numero di concerti.

- `ConcertIncomeTask`: in questo caso non vi è alcuna co-locazione tra task e dati. Viene infatti creato un unico task che viene inviato tramite il metodo `call()` a un qualsiasi nodo server remoto e lì eseguito. Così facendo, avviene un vero e proprio trasferimento di dati attraverso la rete.

In Figura 86 della pagina seguente sono riportati i tempi di esecuzione registrati per ciascuna versione, considerando 500 concerti e 1 milione di biglietti, utilizzando la Macchina 2 e 4 nodi server. Come si può vedere, la versione che non effettua nessuna

```
ignite_test_scenarios-ignite_client-1          | TOTAL PARTITIONS USED: 500
ignite_test_scenarios-ignite_client-1          |
ignite_test_scenarios-ignite_client-1          | +++ TASK-DATA COLOCATION VERSIONS +++
ignite_test_scenarios-ignite_client-1          | >>>    KEY VERSION
ignite_test_scenarios-ignite_client-1          | Output size: 500
ignite_test_scenarios-ignite_client-1          | Sample output: [(Concert0, 107421.40), (Concert1, 101940.98), (Concert10, 111694.13), (Concert100, 100823.30),
 (Concert101, 111481.51), (Concert102, 106030.14), (Concert103, 106678.01), (Concert104, 104729.37), (Concert105, 103813.62), (Concert106, 101779.89)]
ignite_test_scenarios-ignite_client-1          | Execution time: 354209 ms
ignite_test_scenarios-ignite_client-1          |
ignite_test_scenarios-ignite_client-1          | >>>    NOT SCALABLE PARTITION VERSION (PARTITION DEPENDENT)
ignite_test_scenarios-ignite_client-1          | Output size: 500
ignite_test_scenarios-ignite_client-1          | Sample output: [(Concert0, 107421.40), (Concert1, 101940.98), (Concert10, 111694.13), (Concert100, 100823.30),
 (Concert101, 111481.51), (Concert102, 106030.14), (Concert103, 106678.01), (Concert104, 104729.37), (Concert105, 103813.62), (Concert106, 101779.89)]
ignite_test_scenarios-ignite_client-1          | Execution time: 300683 ms
ignite_test_scenarios-ignite_client-1          |
ignite_test_scenarios-ignite_client-1          | >>>    PARTITION VERSION
ignite_test_scenarios-ignite_client-1          | Output size: 500
ignite_test_scenarios-ignite_client-1          | Sample output: [(Concert0, 107421.40), (Concert1, 101940.98), (Concert10, 111694.13), (Concert100, 100823.30),
 (Concert101, 111481.51), (Concert102, 106030.14), (Concert103, 106678.01), (Concert104, 104729.37), (Concert105, 103813.62), (Concert106, 101779.89)]
ignite_test_scenarios-ignite_client-1          | Execution time: 341626 ms
ignite_test_scenarios-ignite_client-1          |
ignite_test_scenarios-ignite_client-1          | +++ WITHOUT TASK-DATA COLOCATION VERSION +++
ignite_test_scenarios-ignite_server-4          | Computation is going to be performed on this node (node ID: 5fc331da-4483-421f-9bc6-ce500bf8f4bf)
ignite_test_scenarios-ignite_client-1          | Output size: 500
ignite_test_scenarios-ignite_client-1          | Sample output: [(Concert0, 107421.40), (Concert1, 101940.98), (Concert10, 111694.13), (Concert100, 100823.30),
 (Concert101, 111481.51), (Concert102, 106030.14), (Concert103, 106678.01), (Concert104, 104729.37), (Concert105, 103813.62), (Concert106, 101779.89)]
ignite_test_scenarios-ignite_client-1          | Execution time: 464640 ms
```

Figura 86: Output dello scenario di test 4.4

co-locazione tra task e dati risulta essere quella meno performante: il maggior tempo è essenzialmente dovuto alla necessità di dover trasferire localmente, sul nodo in cui avviene la computazione, tutti i biglietti distribuiti sul cluster. La versione migliore è invece quella che effettua la co-locazione sulla base delle partizioni nella modalità non scalabile, seguita a breve distanza da quella scalabile. Quest'ultima, infatti, per garantire la correttezza del risultato, è costretta ad utilizzare una modalità di interrogazione che, seppur locale, risulta essere leggermente meno performante rispetto alla scan query utilizzata nel primo caso. Per la stessa ragione, sebbene in questo caso il numero di chiavi della cache `Concert` corrisponda al numero di partizioni, anche la versione basata sulle chiavi ha performato leggermente peggio delle altre versioni co-locate. Per riscontrare quindi le reali differenze di performance tra questa versione e quelle basate sulle partizioni, è stato rieseguito il test considerando questa volta 5 mila concerti e 10 mila biglietti, usando la Macchina 1 con 2 nodi server. Come si può vedere in Figura 87, in questo caso la versione basata sulle chiavi ha impiegato più del doppio rispetto alle altre versioni co-locate a causa del più alto numero di task generati, mentre la versione non scalabile basata sulle partizioni ha prodotto risultati non corretti.

```
ignite_test_scenarios-ignite_client-1  | TOTAL PARTITIONS USED: 1024
ignite_test_scenarios-ignite_client-1  |
ignite_test_scenarios-ignite_client-1  | +++ TASK-DATA COLOCATION VERSIONS +++
ignite_test_scenarios-ignite_client-1  | >>>    KEY VERSION
ignite_test_scenarios-ignite_client-1  | Output size: 5000
ignite_test_scenarios-ignite_client-1  | Sample output: [(Concert0, 159.98), (Concert1, 91.00), (Concert10, 107.48), (Concert100, 159.98), (Concert1000, 125.49),
(Concert1001, 270.96), (Concert1002, 0.00), (Concert1003, 156.48), (Concert1004, 30.99), (Concert1005, 110.98)]
ignite_test_scenarios-ignite_client-1  | Execution time: 57345 ms
ignite_test_scenarios-ignite_client-1  |
ignite_test_scenarios-ignite_client-1  | >>>    NOT SCALABLE PARTITION VERSION (PARTITION DEPENDENT)
ignite_test_scenarios-ignite_client-1  | Output size: 1024
ignite_test_scenarios-ignite_client-1  | Sample output: [(Concert0, 2684.45), (Concert1, 2664.75), (Concert10, 3702.00), (Concert100, 2837.20), (Concert1000, 1505
.88), (Concert1001, 2001.72), (Concert1002, 1367.92), (Concert1003, 1433.84), (Concert1004, 677.84), (Concert1005, 1571.80)]
ignite_test_scenarios-ignite_client-1  | Execution time: 28020 ms
ignite_test_scenarios-ignite_client-1  |
ignite_test_scenarios-ignite_client-1  | >>>    PARTITION VERSION
ignite_test_scenarios-ignite_client-1  | Output size: 5000
ignite_test_scenarios-ignite_client-1  | Sample output: [(Concert0, 159.98), (Concert1, 91.00), (Concert10, 107.48), (Concert100, 159.98), (Concert1000, 125.49),
(Concert1001, 270.96), (Concert1002, 0.00), (Concert1003, 156.48), (Concert1004, 30.99), (Concert1005, 110.98)]
ignite_test_scenarios-ignite_client-1  | Execution time: 23389 ms
```

Figura 87: Output dello scenario di test 4.4

8.2.15 Transazioni ACID (scenario 5.1)

In questo test si utilizzano i meccanismi messi a disposizione da Ignite per implementare le transazioni ACID-compliant. Il test è composto da 1 client e 2 server che conterranno i dati da manipolare. Il client lancia le transazioni e riceve il risultato dai server. Le transazioni vengono gestite in thread separati e lanciate simultaneamente per simulare la concorrenza.

Basandosi sul dominio relativo alla prenotazione online di biglietti di concerti già precedentemente adottato, sono stati inseriti una serie di `Ticket` per simulare l'acquisto concorrente da parte di diverse persone. Sfruttando i meccanismi di lock (automatici in modalità pessimistica), l'obiettivo è quello di ottenere come risultato gli effettivi acquisti dei biglietti e di verificare la corretta modifica dei campi interessati nelle cache, senza che si verifichino interferenze tra le transazioni.

Il test verifica la correttezza delle operazione tramite l'utilizzo di `assertEqual()`. Nel caso in cui queste non producano delle eccezioni, il test è considerato corretto.

8.2.16 Continuous query (scenario 6.1)

In questo test, che mostra il funzionamento delle continuous query in Ignite, viene simulata l'esistenza di un sistema di storage che memorizza i ticket acquistati (in aderenza al dominio già menzionato relativo alla prenotazione di biglietti) e nel quale ne verranno inseriti degli altri in seguito all'attivazione di una continuous query. Attraverso di essa, è possibile specificare una query iniziale e un filtro remoto, con cui andare a reperire i biglietti relativi a uno specifico concerto. Come si vede dall'esempio, Ignite mette a disposizione la sua implementazione di continuous query mediante la classe `ContinuousQuery`. È necessario assegnare un local listener all'istanza della classe prima di eseguirla, tramite il quale si potrà reagire ai nuovi record inseriti e rilevati dalla query.

Osservando l'output in Figura 88, si nota come i record precedentemente presenti vengono estratti e gestiti con una stampa diversa da quelli inseriti dopo l'attivazione della continuous query. In più si vede come tutti i record ottenuti corrispondano alla condizione specificata nella query.

```
ignite_test_scenarios-ignite_client-1  | >>> Test 1 Cache continuous query example started.
ignite_test_scenarios-ignite_client-1  | Queried existing entry [key=3, val=[ ID: 3 - Type: base - UserId: 17 - ConcertId: 1 ]]
ignite_test_scenarios-ignite_client-1  | Queried existing entry [key=12, val=[ ID: 12 - Type: vip - UserId: 19 - ConcertId: 1 ]]
ignite_test_scenarios-ignite_client-1  | Queried existing entry [key=0, val=[ ID: 0 - Type: base - UserId: 1 - ConcertId: 1 ]]
ignite_test_scenarios-ignite_client-1  | Queried existing entry [key=8, val=[ ID: 8 - Type: vip - UserId: 3 - ConcertId: 1 ]]
ignite_test_scenarios-ignite_client-1  | Queried existing entry [key=16, val=[ ID: 16 - Type: vip - UserId: 9 - ConcertId: 1 ]]
ignite_test_scenarios-ignite_client-1  | Queried existing entry [key=18, val=[ ID: 18 - Type: base - UserId: 13 - ConcertId: 1 ]]
ignite_test_scenarios-ignite_client-1  | Updated entry [key=24, val=[ ID: 24 - Type: base - UserId: 20 - ConcertId: 1 ]]
ignite_test_scenarios-ignite_client-1  | Updated entry [key=25, val=[ ID: 25 - Type: base - UserId: 21 - ConcertId: 1 ]]
ignite_test_scenarios-ignite_client-1  | Updated entry [key=26, val=[ ID: 26 - Type: base - UserId: 22 - ConcertId: 1 ]]
ignite_test_scenarios-ignite_client-1  | Updated entry [key=27, val=[ ID: 27 - Type: base - UserId: 23 - ConcertId: 1 ]]
ignite_test_scenarios-ignite_client-1  | Updated entry [key=28, val=[ ID: 28 - Type: base - UserId: 24 - ConcertId: 1 ]]
ignite_test_scenarios-ignite_client-1  | Updated entry [key=29, val=[ ID: 29 - Type: base - UserId: 25 - ConcertId: 1 ]]
ignite_test_scenarios-ignite_client-1  | Updated entry [key=30, val=[ ID: 30 - Type: base - UserId: 26 - ConcertId: 1 ]]
ignite_test_scenarios-ignite_client-1  | Updated entry [key=31, val=[ ID: 31 - Type: base - UserId: 27 - ConcertId: 1 ]]
ignite_test_scenarios-ignite_client-1  | Updated entry [key=32, val=[ ID: 32 - Type: base - UserId: 28 - ConcertId: 1 ]]
ignite_test_scenarios-ignite_client-1  | Updated entry [key=33, val=[ ID: 33 - Type: base - UserId: 29 - ConcertId: 1 ]]
```

Figura 88: Output dello scenario di test 6.1

8.2.17 Data streaming (scenario 7.1)

Per poter effettuare questo test si simulano vari flussi di dati, specifici per funzionalità testata. Alcuni vengono generati in maniera random, con seed fissato, in modo da poter verificare la correttezza dei risultati. Il test è composto da 1 nodo client, che lancia sequenzialmente i test sulle varie funzionalità e ne verifica la correttezza in base all'output, e da 2 nodi server. Le funzionalità prese in esame sono le varie implementazioni di `StreamReceiver`:

- `StreamReceiver`: implementazione classica del receiver che permette di modificare i dati che passano dallo stream prima di essere salvati.
- `StreamTransformer`: implementazione che permette di aggiornare gli elementi esistenti nella cache dello streaming basandosi su elementi precedentemente salvati.
- `StreamVisitor`: implementazione che visita ogni coppia chiave-valore nello stream. Di default, non effettua inserimenti nella cache e quindi necessita dell'utilizzo di metodi di put chiamati esplicitamente.

Il test è suddiviso in tre fasi, ognuna riferita ad una delle tre implementazioni elencate sopra. Tutti i test prevedono uno stream di dati e la sua gestione. La correttezza del test è verificata tramite l'utilizzo di assertion che controllano il valore dei dati processati nelle varie fasi.

8.2.18 Messaging (scenario 8.1)

In questo test viene simulato lo scambio di messaggi e mostrato come un nodo può reagire a tipologie specifiche di messaggi in base al topic e al contenuto. Il test si compone di due fasi: nella prima, viene mostrata la differenza tra l'invio di messaggi non ordinati (Figura 89) e l'invio di quelli con vincolo di ordinamento (Figura 90 pagina seguente).

```
ignite_test_scenarios-ignite_client-1  | >>> Start sending unordered messages.
ignite_test_scenarios-ignite_client-1  | >>> Unordered messages sent.
ignite_test_scenarios-ignite_server-1    Received unordered message [msg=1, fromNodeId=7b4579fb-6d3c-4618-ba0e-517ab0f22139]
ignite_test_scenarios-ignite_server-1    Received unordered message [msg=0, fromNodeId=7b4579fb-6d3c-4618-ba0e-517ab0f22139]
ignite_test_scenarios-ignite_server-1    Received unordered message [msg=5, fromNodeId=7b4579fb-6d3c-4618-ba0e-517ab0f22139]
ignite_test_scenarios-ignite_server-1    Received unordered message [msg=7, fromNodeId=7b4579fb-6d3c-4618-ba0e-517ab0f22139]
ignite_test_scenarios-ignite_server-1    Received unordered message [msg=3, fromNodeId=7b4579fb-6d3c-4618-ba0e-517ab0f22139]
ignite_test_scenarios-ignite_server-1    Received unordered message [msg=2, fromNodeId=7b4579fb-6d3c-4618-ba0e-517ab0f22139]
ignite_test_scenarios-ignite_server-1    Received unordered message [msg=4, fromNodeId=7b4579fb-6d3c-4618-ba0e-517ab0f22139]
ignite_test_scenarios-ignite_server-1    Received unordered message [msg=6, fromNodeId=7b4579fb-6d3c-4618-ba0e-517ab0f22139]
ignite_test_scenarios-ignite_server-1    Received unordered message [msg=9, fromNodeId=7b4579fb-6d3c-4618-ba0e-517ab0f22139]
ignite_test_scenarios-ignite_server-1    Received unordered message [msg=8, fromNodeId=7b4579fb-6d3c-4618-ba0e-517ab0f22139]
```

Figura 89: Output dello scenario di test 8.1 - Messaggi non ordinati

```
ignite_test_scenarios-ignite_client-1  | >>> Start sending ordered messages.
ignite_test_scenarios-ignite_server-1  | Received ordered message [msg=0, fromNodeId=e0bd8e3a-e4eb-4de6-a052-b1533633e993]
ignite_test_scenarios-ignite_server-1  | Received ordered message [msg=1, fromNodeId=e0bd8e3a-e4eb-4de6-a052-b1533633e993]
ignite_test_scenarios-ignite_server-1  | Received ordered message [msg=2, fromNodeId=e0bd8e3a-e4eb-4de6-a052-b1533633e993]
ignite_test_scenarios-ignite_server-1  | Received ordered message [msg=3, fromNodeId=e0bd8e3a-e4eb-4de6-a052-b1533633e993]
ignite_test_scenarios-ignite_server-1  | Received ordered message [msg=4, fromNodeId=e0bd8e3a-e4eb-4de6-a052-b1533633e993]
ignite_test_scenarios-ignite_server-1  | Received ordered message [msg=5, fromNodeId=e0bd8e3a-e4eb-4de6-a052-b1533633e993]
ignite_test_scenarios-ignite_server-1  | Received ordered message [msg=6, fromNodeId=e0bd8e3a-e4eb-4de6-a052-b1533633e993]
ignite_test_scenarios-ignite_server-1  | Received ordered message [msg=7, fromNodeId=e0bd8e3a-e4eb-4de6-a052-b1533633e993]
ignite_test_scenarios-ignite_server-1  | Received ordered message [msg=8, fromNodeId=e0bd8e3a-e4eb-4de6-a052-b1533633e993]
ignite_test_scenarios-ignite_server-1  | Received ordered message [msg=9, fromNodeId=e0bd8e3a-e4eb-4de6-a052-b1533633e993]
ignite_test_scenarios-ignite_client-1  | >>> Ordered messages sent.
ignite_test_scenarios-ignite_client-1  | >>> Check the remote nodes output to see the prints.
ignite_test_scenarios-ignite_client-1  | >>> Waiting for the acknowledgements of the remote nodes.
ignite_test_scenarios-ignite_client-1  | >>> Test 1 ordered-unordered messages completed.
```

Figura 90: Output dello scenario di test 8.1 - Messaggi ordinati

Nella seconda, viene mostrato come si possa automatizzare la gestione della ricezione dei messaggi e delle risposte da mandare in base a quello che si riceve. L'esempio riportato è quello classico nell'ambito della messaggistica, il Ping-Pong. Come si può vedere vedere dalla Figura 91, il nodo client parte con il primo messaggio di "ping" che dà inizio allo scambio, al quale il server risponde con un "pong". Nel caso in cui si esegua il test con più di un solo server, la prima fase del test invierà a tutti i nodi server i messaggi, ordinati e non ordinati. Per quanto riguarda la seconda fase, invece, il client simulerà un numero di partite pari al numero di server.

```
ignite_test_scenarios-ignite_client-1  | >>> Test 2 ping-pong messages simulation.
ignite_test_scenarios-ignite_client-1  | Received message [msg=PONG, sender=6d53ad0a-bffd-48ab-b960-8707f153985f]
ignite_test_scenarios-ignite_server-1  | Received message [msg=PING, sender=e0bd8e3a-e4eb-4de6-a052-b1533633e993]
ignite_test_scenarios-ignite_client-1  | Received message [msg=PONG, sender=6d53ad0a-bffd-48ab-b960-8707f153985f]
ignite_test_scenarios-ignite_server-1  | Received message [msg=PING, sender=e0bd8e3a-e4eb-4de6-a052-b1533633e993]
ignite_test_scenarios-ignite_client-1  | Received message [msg=PONG, sender=6d53ad0a-bffd-48ab-b960-8707f153985f]
ignite_test_scenarios-ignite_server-1  | Received message [msg=PING, sender=e0bd8e3a-e4eb-4de6-a052-b1533633e993]
ignite_test_scenarios-ignite_client-1  | Received message [msg=PONG, sender=6d53ad0a-bffd-48ab-b960-8707f153985f]
ignite_test_scenarios-ignite_server-1  | Received message [msg=PING, sender=e0bd8e3a-e4eb-4de6-a052-b1533633e993]
ignite_test_scenarios-ignite_client-1  | Received message [msg=PONG, sender=6d53ad0a-bffd-48ab-b960-8707f153985f]
ignite_test_scenarios-ignite_server-1  | Received message [msg=PING, sender=e0bd8e3a-e4eb-4de6-a052-b1533633e993]
ignite_test_scenarios-ignite_client-1  | Received message [msg=PONG, sender=6d53ad0a-bffd-48ab-b960-8707f153985f]
ignite_test_scenarios-ignite_server-1  | Received message [msg=PING, sender=e0bd8e3a-e4eb-4de6-a052-b1533633e993]
ignite_test_scenarios-ignite_client-1  | Received message [msg=PONG, sender=6d53ad0a-bffd-48ab-b960-8707f153985f]
ignite_test_scenarios-ignite_server-1  | Received message [msg=PING, sender=e0bd8e3a-e4eb-4de6-a052-b1533633e993]
ignite_test_scenarios-ignite_client-1  | Received message [msg=PONG, sender=6d53ad0a-bffd-48ab-b960-8707f153985f]
ignite_test_scenarios-ignite_server-1  | Received message [msg=PING, sender=e0bd8e3a-e4eb-4de6-a052-b1533633e993]
ignite_test_scenarios-ignite_client-1  | Received message [msg=PONG, sender=6d53ad0a-bffd-48ab-b960-8707f153985f]
ignite_test_scenarios-ignite_server-1  | Received message [msg=PING, sender=e0bd8e3a-e4eb-4de6-a052-b1533633e993]
ignite_test_scenarios-ignite_client-1  | Received message [msg=PONG, sender=6d53ad0a-bffd-48ab-b960-8707f153985f]
ignite_test_scenarios-ignite_server-1  | Received message [msg=STOP, sender=e0bd8e3a-e4eb-4de6-a052-b1533633e993]
ignite_test_scenarios-ignite_server-1  | >>> Test 2 ping-pong messages simulation completed.
```

Figura 91: Output dello scenario di test 8.1 - Ping-Pong

8.2.19 Hadoop acceleration (scenario 9.1)

Lo scopo di questo test è di implementare il meccanismo che permette l'accelerazione di un task MapReduce. Lo scenario di riferimento è quello in cui si ha già a disposizione un cluster Hadoop e si vuole ottenere un miglioramento generale delle prestazioni senza dover migrare verso una nuova piattaforma. Con Ignite è infatti possibile innestare un layer in-memory nell'architettura tradizionale di Hadoop e sfruttarlo a proprio vantaggio.

Il task preso in considerazione è quello già descritto nello scenario 4.1 e nello scenario 4.2 riguardante il calcolo del tf-idf di ogni parola contenuta nella *Divina Commedia*, qui re-implementato per usare l'API della classica implementazione del paradigma MapReduce fornita da Hadoop.

Per eseguire il deploy del task sul cluster Hadoop, composto per semplicità da un singolo nodo, vengono dapprima lanciati tutti i servizi Hadoop necessari, tra cui HDFS su cui vengono caricati i file di input e scritti i vari file di output.

Considerata la diversa modalità di esecuzione rispetto ai precedenti test, sono stati eccezionalmente predisposti un file di Compose e un Dockerfile appositi (rispettivamente `docker-compose.9.1.yaml` e `Dockerfile9.1`), che gestiscono il lancio di un singolo container nel quale viene esguito lo script che provvede a scaricare ed installare tutte le risorse necessarie, a configurare il cluster Hadoop in maniera opportuna e infine ad eseguire il task.

Nello specifico, il comando da lanciare in questo caso è il seguente:

```
1 docker-compose --env-file .env.9.1 -f docker-compose.9.1.yaml run hadoop_node
```

Per avviare il test in modalità accelerata, è sufficiente impostare la variabile `MODE`, contenuta nell'apposito file `.env`, con il valore `acceleration`. L'esecuzione impiegherà qualche minuto e ad un certo punto sarà necessario digitare una doppia conferma (`yes`) richiesta per l'avvio di HDFS.

```
        .
        .
        .
+++ STARTING COMPUTING MAP-REDUCE TASK WITH HADOOP +++
23/04/14 07:27:31 INFO Configuration.deprecation: session.id is deprecated. Instead, use dfs.metrics.session-id
23/04/14 07:27:31 INFO jvm.JvmMetrics: Initializing JVM Metrics with processName=JobTracker, sessionId=
23/04/14 07:27:31 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface and execute your application with ToolRunner to remedy this.
23/04/14 07:27:32 INFO input.FileInputFormat: Total input paths to process : 1
23/04/14 07:27:32 INFO input.FileInputFormat: Total input paths to process : 1
23/04/14 07:27:32 INFO input.FileInputFormat: Total input paths to process : 1
23/04/14 07:27:32 INFO mapreduce.JobSubmitter: number of splits:3
23/04/14 07:27:33 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_local1497886668_0001
23/04/14 07:27:33 INFO mapreduce.Job: The url to track the job: http://localhost:8080/
23/04/14 07:27:33 INFO mapreduce.Job: Running job: job_local1497886668_0001
        .
        .
        .
real    0m49.107s
user    0m33.106s
sys     0m34.346s
+++ TF-IDF TASK OUTPUT +++
a,inferno       0.0
a,paradiso      0.0
a,purgatorio    0.0
ab,inferno      1.4738246523944721E-5
abate,paradiso  5.538505977721622E-6
abate,purgatorio        1.08721797336264 79E-5
abbaglia,inferno        1.4738246523944721E-5
abbandona,paradiso      1.5006644483854264E-5
abbandonai,inferno      1.4738246523944721E-5
abbandonati,purgatorio  1.4729146874931697E-5
abbandonato,purgatorio  1.4729146874931697E-5
abbandonò,inferno       1.4738246523944721E-5
abbarbaglio,paradiso    1.5006644483854264E-5
abbarbicata,inferno     1.4738246523944721E-5
        .
        .
        .
```

Figura 92: Output dello scenario di test 9.1 - modalità classica

In Figura 92 e in Figura 93 della pagina seguente sono mostrati gli snippet delle parti più significative dell'output del test, eseguito rispettivamente in modalità classica e in modalità accelerata. Quest'ultima si differenzia per la presenza di un nodo Ignite,

responsabile dell'accelerazione. Come si può notare, non si è riscontrata una reale differenza nei due casi. Ciò può essere dovuto al fatto di aver usato un singolo nodo Hadoop o dalla natura stessa del task. Mettere in campo ulteriori ottimizzazioni, quali l'uso di IGFS e della cache di secondo livello, potrebbe far emergere le differenze. Non da ultimo, sebbene sia ancora pubblicizzata dalle fonti ufficiali, l'accelerazione di Hadoop risulta essere una feature dismessa e non più supportata dalle recenti versioni di Ignite.

```
          .
          .
          .
+++ STARTING AN IGNITE NODE +++
[07:53:08]    __________  ________________
[07:53:08]   /  _/ ___/ |/ /  _/_  __/ __/
[07:53:08]  _/ // (7 7    // /  / / / _/
[07:53:08] /___/\___/_/|_/___/ /_/ /___/
[07:53:08]
[07:53:08] ver. 1.6.0#20160518-sha1:0b22c45b
[07:53:08] 2016 Copyright(C) Apache Software Foundation
[07:53:08]
[07:53:08] Ignite documentation: http://ignite.apache.org
[07:53:08]
[07:53:08] Quiet mode.
[07:53:08]   ^-- Logging to file '/apache-ignite-fabric-1.6.0-bin/work/log/ignite-7c4d2dbf.0.log'
[07:53:08]   ^-- To see **FULL** console log here add -DIGNITE_QUIET=false or "-v" to ignite.{sh|bat}
[07:53:08]
[07:53:08] OS: Linux 5.15.49-linuxkit amd64
[07:53:08] VM information: OpenJDK Runtime Environment 1.8.0_362-8u362-ga-0ubuntu1~22.04-b09 Private Build OpenJDK 64-Bit Server VM 25.362-b09
[07:53:08] Configured plugins:
[07:53:08]   ^-- None
[07:53:08]
[07:53:08] Security status [authentication=off, tls/ssl=off]
[07:53:09] HADOOP_HOME is set to /hadoop-2.7.2
[07:53:10] To start Console Management & Monitoring run ignitevisorcmd.{sh|bat}
[07:53:10]
[07:53:10] Ignite node started OK (id=7c4d2dbf)
[07:53:10] Topology snapshot [ver=1, servers=1, clients=0, CPUs=4, heap=1.0GB]
+++ STARTING COMPUTING MAP-REDUCE TASK WITH IGNITE HADOOP ACCELERATION +++
Apr 14, 2023 7:53:20 AM org.apache.ignite.internal.client.impl.connection.GridClientNioTcpConnection <init>
INFO: Client TCP connection established: localhost/127.0.0.1:11211
Apr 14, 2023 7:53:20 AM org.apache.ignite.internal.client.impl.GridClientImpl <init>
INFO: Client started [id=f5122e8a-ad48-4487-a300-31a7716cac93, protocol=TCP]
23/04/14 07:53:21 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface and execute your application with ToolRunner to remedy this.
23/04/14 07:53:22 INFO input.FileInputFormat: Total input paths to process : 1
23/04/14 07:53:22 INFO input.FileInputFormat: Total input paths to process : 1
23/04/14 07:53:22 INFO input.FileInputFormat: Total input paths to process : 1
23/04/14 07:53:22 INFO mapreduce.JobSubmitter: number of splits:3
23/04/14 07:53:22 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_7c4d2dbf-32f0-4859-809f-5b025219ce99_0001
log4j:WARN No appenders could be found for logger (org.apache.hadoop.security.authentication.util.KerberosName).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
23/04/14 07:53:26 INFO mapreduce.Job: The url to track the job: N/A
23/04/14 07:53:26 INFO mapreduce.Job: Running job: job_7c4d2dbf-32f0-4859-809f-5b025219ce99_0001
          .
          .
          .
real    0m52.118s
user    0m15.839s
sys     0m16.936s
+++ TF-IDF TASK OUTPUT +++
a,inferno       0.0
a,paradiso      0.0
a,purgatorio    0.0
ab,inferno      1.4738246523944721E-5
abate,paradiso  5.538505977721622E-6
abate,purgatorio        1.0872179733626479E-5
abbaglia,inferno        1.4738246523944721E-5
abbandona,paradiso      1.5006644483854264E-5
abbandonai,inferno      1.4738246523944721E-5
abbandonati,purgatorio  1.4729146874931697E-5
abbandonato,purgatorio  1.4729146874931697E-5
abbandonò,inferno       1.4738246523944721E-5
abbarbaglio,paradiso    1.5006644483854264E-5
abbarbicata,inferno     1.4738246523944721E-5
          .
          .
          .
```

Figura 93: Output dello scenario di test 9.1 - modalità accelerata

9 Conclusioni

Con questo progetto si è voluto esplorare ed analizzare a fondo quella che sembra essere una delle tecnologie più promettenti nell'ambito della computazione distribuita. Il principale punto di forza di Apache Ignite consiste nel fornire in un'unica piattaforma uno strumento in grado di supportare contemporaneamente la computazione distribuita e un sistema di storage distribuito, affidandosi ad un cluster di nodi, distinguendosi dalla maggior parte dei framework in circolazione che spesso si focalizzano invece su uno dei due aspetti.

Questa natura "general purpose" di Ignite, derivante soprattutto dal fatto che essa fornisce un'ampia suite di funzionalità che spaziano dalla memorizzazione, computazione e interrogazione distribuita al messaging, passando per il Machine Learning, l'ha resa uno strumento popolare e adatto soprattutto per l'ambito aziendale, in realtà medio-grandi che hanno avuto necessità nel tempo di adattare i propri processi a nuovi standard operativi o che si sono affidati a questa tecnologia sin dall'inizio, modellando e implementando ex-novo i propri servizi sulla base di essa. Nonostante ciò, non mancano esempi di scenari di ricerca che hanno potuto beneficiare di Ignite e trarre vantaggio dalla sua natura prettamente in-memory per velocizzare e parallelizzare i propri esperimenti. La possibilità di mantenere tutti i dati in memoria, in un unico posto, e aggiornarli in tempo reale grazie alla feature di data streaming, fa sì che questi siano sempre aggiornati ed evita di dover procedere con costose operazioni periodiche di ETL. Per tutti questi motivi, Ignite può essere impiegato con successo per l'implementazione di sistemi di continuous learning.

Grazie agli aspetti architetturali e ai meccanismi implementati da Ignite, descritti e approfonditi nel corso della trattazione, la tecnologia è in grado di offrire le seguenti **proprietà distribuite**:

- **elasticità**: il cluster può essere scalato orizzontalmente in maniera potenzialmente illimitata, aggiungendo (o rimuovendo) nodi server in base al mutare delle condizioni di workload. I meccanismi automatici di data rebalancing e load balancing assicurano rispettivamente che la distribuzione dei dati e dei task computazionali seguano i cambiamenti di topologia.
- **high availability**: grazie alla possibilità di impostare copie di backup per ciascuna partizione fino a replicare interamente i dati su ciascun nodo.
- **fault tolerance**: grazie alla presenza dei backup e alla possibilità di sfruttare diverse modalità di persistenza, che vanno ad aggiungersi alle copie primarie dei dati in cache.
- **safety**: viste le garanzie di fault tolerance che offre, Ignite può essere catalogato come *dependable system*, ovvero un sistema "affidabile". Una tipica proprietà di questo genere di sistemi è appunto la safety, che in tal caso garantisce che il sistema continui ad operare correttamente se un nodo si disconnette temporaneamente, per un breve periodo di tempo, dal resto del cluster, senza che venga attivato il meccanismo di data rebalancing.

- **strong persistence**: quando è attiva una modalità di persistenza, nativa o esterna, che va a replicare i dati anche su disco.
- in relazione al teorema CAP, Ignite può comportarsi sia come sistema **AP**, quando si trova in modalità atomica e quindi la precedenza viene data alla disponibilità, sia come sistema **CP**, quando si trova in modalità transazionale e viene data priorità alla consistenza dei dati.

In base a quanto approfondito e sperimentato attraverso l'implementazione dei diversi scenari di test, i principali **vantaggi** e punti di forza di Ignite sono i seguenti:

- architettura in-memory, che sfrutta la RAM sia per la computazione che per la memorizzazione dei dati.
- architettura shared-nothing, che fa sì che tutti i nodi del cluster siano indipendenti dal punto di vista delle risorse e della coordinazione. In questo modo, non esistono componenti bloccanti da cui gli altri dipendono e che possono influenzare in maniera negativa la resa del sistema.
- strettamente legato al punto precedente, l'assenza di un coordinatore distingue Ignite da altre tecnologie simili, come Hadoop, in cui viene adottato un pattern di tipo master-slave. In Ignite, tutti i nodi server vengono trattati come entità di pari livello, con le medesime funzionalità e lo stesso comportamento.
- ogni nodo server funge sia da compute node che da data node, in maniera trasparente.
- possibilità di scaling orizzontale illimitato.
- meccanismo di discovery automatico dei nodi: ogni nuovo nodo Ignite entra a far parte del cluster in maniera autonoma.
- l'archiviazione a più livelli, che permette di sfruttare all'occorrenza anche il disco, offre un'incrementata garanzia di consistenza che riduce sensibilmente la possibilità di perdita di dati, grazie ai meccanismi di WAL e checkpointing.
- usato come layer intermedio per incrementare le performance di sistemi esistenti, in modalità external storage, è in grado di aggiungere pieno supporto a SQL, utile nel caso in cui il DB del sistema primario non lo preveda, senza peraltro dover migrare i dati. Se ad esempio il database primario è Cassandra, tale integrazione permetterà di eseguire join (distribuiti, co-locati o non) che nel linguaggio CQL, usato da questo tipo di DB, non sono previsti.
- il data model su cui si basa Ignite offre due diverse rappresentazioni logiche dei dati (*i.e.* chiave-valore e SQL Table) che possono venire usate indifferentemente e congiuntamente. La SQL Table può agevolare gli utenti che provengono dal mondo SQL.

- essendo basato sull'in-memory computing, il framework è particolarmente adatto per un uso iterativo e interattivo, a differenza ad esempio di Hadoop che è più indicato per il batch processing.

- come sperimentato nello scenario 4.3, Ignite offre un meccanismo di automatic job failover in grado di sottomettere nuovamente il task che era in esecuzione al momento della failure su un altro nodo.

Gli **svantaggi** e le criticità che si hanno avuto modo di osservare sono le seguenti:

- la topologia ad anello adottata normalmente da Ignite diventa limitante in presenza di migliaia di nodi. In tale situazione è quindi necessario passare allo Zookeeper Discovery, che richiede tuttavia la configurazione e la gestione di un cluster aggiuntivo, con conseguente overhead di gestione.

- per quanto riguarda il data model chiave-valore, nel caso in cui una cache memorizzi come valore un oggetto complesso e si voglia andare ad aggiornare un singolo campo di quell'oggetto, è necessario fare la get del valore, modificare il campo e sovrascrivere il dato presente sulla cache. Questo procedimento può essere molto costoso per scenari che prevedono di modificare molto spesso un campo di un oggetto molto grande. Si tratta tuttavia di un problema generale, individuabile in tutte le tecnologie che adottano tale data model e non è quindi una limitazione specifica di Ignite.

- come riscontrato nello scenario 1.2, la modalità di caching on-heap non sempre è più performante di quella off-heap. All'aumentare del numero di entry memorizzate in cache, infatti, si assiste ad un progressivo deterioramento delle performance in lettura. Tale comportamento può trovare spiegazione nell'overhead del garbage collector, che aumenta la sua attività all'aumentare dello spazio heap occupato. La documentazione ufficiale non è chiara al riguardo: la modalità on-heap non è, quindi, da preferire a priori quando si fanno molte letture, ma piuttosto è da circoscrivere alle situazioni in cui si fanno molte letture ma su una piccola porzione dei dati, tale da non comportare un'intesa attività di garbage collection. Per quanto riguarda le performance in scrittura, infine, la modalità on-heap risulta sempre la peggiore, dal momento che deve memorizzare i dati sia nella regione off-heap che nello heap.

- necessità di pre-caricare tutti i dati in RAM prima di poter usare la SQL API per eseguire delle query. La corretta modalità attraverso cui eseguire con successo tale operazione non è indicata nella documentazione ufficiale.

- implementare in Ignite un task con il paradigma MapReduce richiede una gestione più verbosa del codice nella gestione dei vari job, nonostante le classi di utility fornite per agevolarne l'implementazione. Al contrario, le tradizionali implementazioni del paradigma tramite Hadoop sono più snelle, soprattutto in relazione alla tipizzazione predefinita dei dati in input e in output ai job. Inoltre, nella versione

di Ignite è necessario gestire esplicitamente come partizionare i job di map sulla base dei dati di input, mentre la versione di Hadoop gestisce tutto in maniera trasparente.

- il meccanismo di checkpointing, da abilitare espressamente e da usare congiuntamente a quello di automatic job failover, non sempre porta a un effettivo vantaggio in caso di failure. Se infatti la computazione non è abbastanza complessa, oppure se i risultati intermedi occupano tanto spazio, può succedere che i tempi necessari al salvataggio e al successivo caricamento del checkpoint superino quelli della computazione in sé, come riscontrato per lo scenario 4.2.

- la task-data colocation è da abilitare esplicitamente e richiede abbastanza codice per la sua gestione, soprattutto se si vuole abilitare anche la co-locazione tra cache per ottenere la versione più efficiente basata sulle partizioni, a differenza di altri strumenti come Hadoop in cui invece viene applicata di default. Di base, quindi, Ignite non sfrutta la data locality e per le computazioni distribuite vengono in realtà spostati i dati, a meno che questi non siano replicati interamente su ogni nodo o non si usi la SQL API, mediante cui le query vengono inviate direttamente ai nodi in cui si trovano i dati di interesse.

- il supporto completo lo si ha solo con le versioni rilasciate a pagamento da GridGain, le quali offrono anche una serie di feature aggiuntive, come la possibilità di distribuire il cluster su più datacenter e maggiori garanzie in relazione alla privacy, che spesso sono necessarie per i contesti aziendali in cui Ignite viene più diffusamente sfruttato.

Infine, si conclude dicendo che nell'implementazione degli scenari di test, si sono incontrate le seguenti **difficoltà**:

- sebbene ad un primo sguardo la documentazione offerta dal sito ufficiale possa sembrare ricca e completa, si sono avute diverse difficoltà nell'implementare alcuni scenari e nel farli funzionare nel modo corretto. Infatti, certi aspetti delle varie feature non sono argomentati a dovere e in alcuni casi essi fanno riferimento a versioni precedenti del framework che funzionano in maniera leggermente diversa.

- eseguire i vari scenari di test è stato spesso molto problematico per via delle limitate risorse a disposizione, considerata anche la modalità di deployment tramite container adottata, che da un lato assicura la riproducibiltà dei test su un qualsiasi host ma dall'altro rallenta notevolmente l'esecuzione. Ignite è infatti pensato per essere eseguito su delle macchine unicamente adibite allo scopo.

- impossibilità di testare alcune feature chiave di Ignite (*e.g.* Zookeeper Discovery), per via della mancanza di un'infrastruttura hardware idonea.

Bibliografia

[1] The Apache Software Foundation. Apache Ignite: Overview, . URL https://ignite.apache.org. Accessed: September, 2022.

[2] The Apache Software Foundation. Apache Ignite: Documentation, . URL https://ignite.apache.org/docs/latest/. Accessed: September, 2022.

[3] The Apache Software Foundation. Apache ZooKeeper: Overview, . URL https://zookeeper.apache.org. Accessed: September, 2022.

[4] The Apache Software Foundation. Apache Cassandra: Overview, . URL https://cassandra.apache.org/_/index.html. Accessed: November, 2022.

[5] The Oracle Corporation. Java Caching API - JSR107 specification. URL https://docs.google.com/document/d/1ijduF_tmHvBaUS7VBBU2ZN8_eEBiFaXXg9OI0_ZxCrA/edit. Accessed: November, 2022.

[6] VMware Inc. Xml Schema-based Configuration, . URL https://docs.spring.io/spring-framework/docs/4.2.x/spring-framework-reference/html/xsd-configuration.html. Accessed: November, 2022.

[7] Enlyft. Companies using Apache Ignite. URL https://enlyft.com/tech/products/apache-ignite. Accessed: December, 2022.

[8] The Apache Software Foundation. Business Use Cases, . URL https://ignite.apache.org/use-cases/provenusecases.html. Accessed: December, 2022.

[9] GridGain Systems Inc. Customer Case Studies, . URL https://www.gridgain.com/experience/case-studies. Accessed: December, 2022.

[10] R. Bansod, S. Kadarkar, R. Virk, M. Raval, R. Rashinkar, and M. Nambiar. High Performance Distributed In-Memory Architectures For Trade Surveillance System. 2018. URL https://doi.org/10.1109/ISPDC2018.2018.00023.

[11] B. Jena and S. K. Sahoo. A Cloud Native SOS Alert System Model Using Distributed Data Grid and Distributed Messaging Platform. 2022. URL https://doi.org/10.1007/978-981-16-9873-6_5.

[12] S. Gorsky, A. Edelev, and A. Feoktistov. Data Processing in Problem-Solving of Energy System Vulnerability Based on In-memory Data Grid. 2022. URL https://doi.org/10.1007/978-3-030-97020-8_25.

[13] A. H. Sojoodi, M. S. Beni, and F. Khunjush. Ignite-GPU: a GPU-enabled in-memory computing architecture on clusters. 2020. URL https://doi.org/10.1007/s11227-020-03390-z.

[14] M. S. Beni, A. H. Sojoodi, and F. Khunjush. A GPU-Enabled Extension for Apache Ignite to Facilitate Running Genetic Algorithms. 2020. URL https://doi.org/10.1109/CADS50570.2020.9211857.

[15] M. M. Alam, S. Ray, and V. C. Bhavsar. A Performance Study of Big Spatial Data Systems. 2018. URL https://doi.org/10.1145/3282834.3282841.

[16] C. S. Stan, A. E. Pandelica, V. A. Zamfir, R. G. Stan, and C. Negru. Apache Spark and Apache Ignite Performance Analysis. 2019. URL https://doi.org/10.1109/CSCS.2019.00129.

[17] Shamim Ahmed Bhuiyan, Michael Zheludkov, and Timur Isachenko. *High Performance in-memory computing with Apache Ignite*. Leanpub, 2017.

[18] Yakov Zhdanov. Apache Ignite - Design Documents. URL https://cwiki.apache.org/confluence/display/IGNITE/Design+Documents. Accessed: November, 2022.