Università degli Studi di Modena e Reggio Emilia

Dipartimento di Ingegneria "Enzo Ferrari"

Corso di Laurea in Ingegneria Informatica

Studio comparativo sulle performance di SQL Server, PostgreSQL, Oracle XE e MySQL

Relatore:

Prof.ssa Sonia Bergamaschi

Candidato:

Valentina Costi

Anno Accademico 2015-2016

SOMMARIO

INDICE DELLE FIGURE

1 INTRODUZIONE

DBMS, acronimo di Database Management System, è un sistema software che permette di creare, memorizzare, manipolare, aggiornare, amministrare e interrogare database, ponendosi come interfaccia tra i dati immagazzinati nel database stesso e l'utente o i programmi applicativi. Un DBMS in particolare consente elaborazioni concorrenti, mantiene la consistenza delle informazioni e permette indipendenza del software dall'organizzazione fisica e logica delle strutture dati [1].
Al giorno d'oggi i DBMS rivestono un ruolo fondamentale in numerose applicazioni informatiche, inerenti una vasta gamma di aspetti e settori della vita quotidiana, che fanno ampiamente uso di database interni come repository per memorizzare e gestire i propri dati; tra queste si trovano, per citarne alcune macro-categorie, applicazioni contabili, per la gestione finanziaria e bancaria, applicazioni CRM[1] e applicazioni web dinamiche, sfruttate ad esempio da molti retail store online.

Sono disponibili sul mercato differenti tipologie e versioni di DBMS, sviluppate da diverse aziende, che si distinguono tra loro per specifiche e funzionalità anche ampiamente differenziate, per esempio in termini di organizzazione interna dei dati, che ne influenzano la flessibilità e la velocità nel reperimento delle informazioni ricercate.

La presente tesi si prefigge lo scopo di valutare e comparare le performance di quattro tra i più conosciuti DBMS: Microsoft SQL Server, PostgreSQL, MySQL e Oracle XE.
Lo studio è basato sull'analisi di alcuni dati e aspetti inerenti la valutazione delle prestazioni, quali i tempi di esecuzione, l'utilizzo di CPU e il numero di threads impiegati per portare a termine le interrogazioni immesse, sia in assenza che in presenza di indici appositamente formulati che comportino incrementi delle prestazioni.
La valutazione delle performance è stata condotta formulando query basate su più operazioni di Full Outer Join successive, secondariamente ridotte a Left/Right Outer Join mediante l'imposizione di condizioni di filtro. La scelta è ricaduta su tale tipologia di operatori in quanto si tratta di una tra le più onerose, pesanti e complesse nella propria esecuzione.
Per approfondire i risultati ottenuti e comprendere più nel dettaglio i diversi approcci adottati e le metodologie impiegate, sono stati inoltre analizzati i piani di esecuzione prodotti contestualmente alle query eseguite.

1.1 ORGANIZZAZIONE DELLA TESI

Il Capitolo 2 consiste in una sintetica descrizione dei principi base secondo i quali i DBMS processano le interrogazioni; in particolare l'attenzione viene focalizzata sul componente Query Optimizer responsabile della produzione dei piani di esecuzione. Vengono quindi fornite le nozioni basilari per comprendere gli aspetti dei query plan più salienti, indagati poi con maggiore attenzione.

[1] CRM è l'acronimo di Customer Relationship Management; si tratta di un sistema in cui è cruciale l'attenzione al cliente, legato al concetto di fidelizzazione dei clienti stessi; vengono impiegati database per la gestione delle vendite, marketing e per il supporto delle relazioni tra l'azienda e i suoi clienti.

Il Capitolo 3 spiega brevemente le motivazioni che hanno condotto alla scelta del Full Outer Join come operatore principale impiegato nella formulazione delle query.
Nel Capitolo 4 si offrono delle sintetiche descrizioni dei quattro DBMS coinvolti nello studio, esponendo le principali funzionalità che caratterizzano ognuno di essi, in modo da fornire al lettore un quadro generale più completo entro il quale collocare le successive osservazioni.
Il Capitolo 5 spiega le caratteristiche, rilevanti per lo studio, della piattaforma impiegata per eseguire i test e gli strumenti utilizzati per il reperimento delle informazioni che permettono di realizzare le analisi comparative.
Il Capitolo 6 descrive come è stata ottenuta la base di dati sulla quale si concentrano i test svolti, fornendo una visione generale della sua struttura e delle relazioni che la costituiscono. Vi è inoltre esposta la metodologia impiegata per condurre le analisi.
Il Capitolo 7 contiene il cuore dello studio, in quanto riporta i piani di esecuzione ritenuti più rilevanti ai fini dell'analisi, con i dati ad essi associati e le considerazioni principali in merito allo studio e alla comparazione delle performance. È strutturato seguendo la suddivisione operata nella conduzione dello studio, ripartendo cioè le informazioni e le osservazioni nei quattro gruppi creati per la sottomissione delle query, di cui si parlerà più nel dettaglio successivamente.
Infine nel Capitolo 8 si possono consultare i principali dati ricavati, esposti in forma di tabelle e grafici, per una visione immediata dei confronti eseguiti.

2 QUERY OPTIMIZER E PIANI DI ESECUZIONE

Il percorso seguito da una query all'interno di un DBMS, dalla sua formulazione al momento in cui è generata la risposta, è costituito da 4 passaggi fondamentali [2]:

- *Query Parser*: controlla la correttezza della query e la traduce in specifiche istruzioni interne, creando data structure che altre routines possono processare;
- *Query Optimizer*: determina il metodo maggiormente efficiente secondo il quale accedere ai dati richiesti, o, in generale, eseguire la query immessa;
- *Code Generator*: traduce il piano prodotto e scelto nel passaggio precedente in chiamate al query processor;
- *Query Processor*: esegue la query.

L'elemento su cui lo studio svolto concentra un maggiore interesse è costituito dal Query Optimizer, in quanto, più delle altre fasi sopra riportate, influisce e determina i risultati in termini di prestazioni.
Il suo scopo infatti, come già accennato, consiste nel determinare la strategia più efficiente per l'esecuzione di una data query, ovvero scegliere la sequenza di operazioni che permetta il reperimento delle informazioni richieste nel minor tempo possibile, basando tale scelta su molteplici fattori, quali la cardinalità delle relazioni coinvolte, la densità delle pagine, la possibile presenza di indici disponibili ed eventuali statistiche impiegabili per reagire in maniera intelligente alle istruzioni ricevute. Tali informazioni rendono l'optimizer in grado di ragionare in maniera più accurata circa la strada da intraprendere per rispondere a quanto chiesto.
Il suo ruolo si concretizza nella produzione di diversi piani di esecuzione. Il piano ottimale è

rappresentato da quello con il minor costo complessivo associato, valore determinato da fattori quali CPU time e operazioni di I/O necessarie.
Tale ricerca può risultare particolarmente complessa e a sua volta costosa in termini di tempo, per cui è spesso ricercato il piano di esecuzione che approssimi l'ottimo, offrendo quindi in un tempo ragionevole un piano "buono a sufficienza".
La scelta fondamentale per un DBMS è quindi identificare il miglior equilibrio contestualmente al trade off tra la qualità del query plan prescelto e la quantità di tempo impiegato per individuarlo; le diverse modalità di bilanciamento dei due aspetti determinano prestazioni e qualità differenti tra DBMS distinti.
Le decisioni principali che il Query Optimizer deve effettuare sono relative a tre elementi chiave:

1. quali Scan methods impiegare: Seek o Scan su indici o tabelle;
2. quali algoritmi di join eseguire: Sort Merge, Hash Join o Nested Loop Join;
3. ordine dei join.

Lo studio condotto ha focalizzato quindi la propria attenzione sui diversi comportamenti adottati dai Query Optimizer dei DBMS in relazione ai punti esposti, approfondendo le osservazioni in merito tramite un'analisi più dettagliata dei piani di esecuzione prodotti, che hanno messo in luce e risaltato maggiormente le differenze strategiche che intercorrono tra i DBMS scelti per l'analisi, e che ne esplicano le differenze prestazionali.
Quindi i piani di esecuzione, prodotti dai Query Optimizer, sono stati sottoposti ad un'attenta analisi perché costituiscono la sorgente di tutte le informazioni per la comprensione dei risultati finali – tempi di esecuzione registrati, numero di threads coinvolti e utilizzo CPU – in relazione alle scelte del singolo DBMS in merito ai tre punti sopra riportati.

3 FULL OUTER JOIN

L'operazione di Full Outer Join permette la restituzione delle righe di entrambe le tabelle o viste specificate nella clausola FROM e coinvolte nel join stesso, a condizione che tali righe soddisfino le eventuali condizioni di ricerca espresse mediante clausola WHERE o HAVING [3].
Tramite tale operatore vengono quindi incluse nei risultati anche le righe prive di corrispondenza, rappresentando le informazioni eventualmente mancanti tramite valori NULL.

La natura e le funzionalità stesse di tale operatore lo identificano come un candidato ideale per essere impiegato nell'ambito del performance testing, in quanto, essendo un join, rientra nella categoria di operazioni maggiormente critiche per un DBMS, e inoltre comporta *result set* di dimensioni anche consistentemente superiori rispetto quelli ottenibili mediante Inner Join.
Tale aspetto si traduce necessariamente nell'esigenza da parte dei DBMS di adottare differenti tecniche per l'ottimizzazione dell'elaborazione dei numerosi dati ritornati, al fine di produrre i risultati richiesti nel minor tempo possibile e mantenere dei tempi di esecuzione competitivi.
L'impiego di appositi accorgimenti e strategie può infatti determinare un divario prestazionale considerevole rispetto al loro mancato utilizzo.
Inoltre, rappresentando il Full Outer Join un'operazione complessa, essa più di altre rappresenta

un valido strumento per un confronto più completo dei piani di esecuzione prodotti, mettendo in luce, oltre l'eventuale presenza di strategie di ottimizzazione, una sequenza di operazioni logiche più ampia per il recupero delle informazioni richieste.

Le query formulate per lo studio presentano pertanto la medesima struttura, incentrata sulla realizzazione di due operazioni consecutive di Full Outer Join, che coinvolgono tre relazioni distinte.
Tale formulazione permette di incrementare ulteriormente la complessità delle interrogazioni, e di conseguenza quella dei piani di esecuzione contestualmente prodotti, rendendo maggiormente marcate le differenze in termini di prestazioni, dettate dalle diverse scelte strategiche condotte dai DBMS.

4 DBMS OGGETTO DI STUDIO

Lo studio è stato condotto su quattro tra i più noti DBMS: MS SQL Server 2008, Oracle XE, MySQL 5.7 e PostgreSQL 9.5.
La presente sezione ne presenta le caratteristiche e funzionalità principali.
È importante sottolineare fin da principio la presenza di un importante elemento che configura una ripartizione dei quattro DBMS citati in due gruppi distinti: da una parte infatti si colloca MS SQL Server 2008, unico dei considerati che richiede una licenza a pagamento per l'utilizzo, mentre dall'altra PostgreSQL 9.5, Oracle XE e MySQL 5.7 sono tutti e tre open source.
Si tratta ovviamente di un aspetto di notevole importanza, in quanto è lecito nutrire delle aspettative maggiori nel caso di DBMS a pagamento, dai quali si richiedono risultati prestazionali superiori, resi possibili anche dai maggiori introiti tramite i quali finanziare lo sviluppo più ampio del proprio prodotto.

4.1 MS SQL SERVER 2008

Microsoft SQL Server è un RDBMS[2] prodotto da Microsoft.
Il linguaggio primario utilizzato per le query è Transact-SQL (T-SQL), un'estensione del linguaggio SQL standard sviluppato da Microsoft e Sybase [3], che implementa features aggiuntive, tra le quali il controllo delle transazioni, la possibilità di definire variabili locali, funzioni per manipolazione di stringhe, date ed espressioni matematiche e la gestione delle eccezioni e degli errori.
Tra le funzionalità previste, si cita il supporto al clustering, che permette di distribuire il carico di lavoro tra una pluralità di server identicamente configurati, e il supporto al database mirroring [4], che permette di aumentare la disponibilità della base di dati, gestendo due copie di un singolo database solitamente ubicati in posizioni diverse.
Prevede inoltre partizione orizzontale, distribuendo le tabelle tra diversi file. [5]
È supportato unicamente da sistema operativo Windows.

[2] RDBMS, acronimo di *Relational DataBase Management System*, è un DBMS basato sul modello relazionale.
[3] Impresa che produce software per la gestione e l'analisi di informazioni in database relazionali. Sito ufficiale.

4.2 POSTGRESQL 9.5

PostgreSQL è un RDBMS a oggetti (ORDBMS) [4], disponibile con licenza libera.
Offre supporto completo per le chiavi esterne, viste, triggers e stored procedures; la sua implementazione del linguaggio SQL è fortemente conforme allo standard ANSI-SQL 2008, quindi, tra gli altri aspetti, include gran parte dei tipi di dato previsti da SQL 2008 e inoltre supporta la memorizzazione di oggetti binari di grandi dimensioni, tra i quali immagini, suoni o video.
PostgreSQL vanta anche funzionalità sofisticate quali Multi-Version Concurrency Control [5], transazioni innestate, point in time recovery [6] e repliche asincrone.
Infine, un altro importante aspetto di vanto per PostgreSQL è l'elevata personalizzazione, in quanto può eseguire stored procedures scritte in differenti linguaggi di programmazione; si possono caricare come librerie triggers e stored procedures scritte in C, garantendo flessibilità nell'estensione delle capacità di tale strumento.
E' supportato dai principali sistemi operativi, inclusi Linux, Unix, Windows e OS X.

4.3 MYSQL 5.7

MySQL è uno dei RDBMS open source più conosciuti a livello globale, da gennaio 2010 acquisita da Oracle Corporation[7]. Alcune delle più grandi organizzazioni mondiali, tra le quali Facebook, Google, Adobe e Alcatel Lucent incorporano MySQL per la gestione dei loro siti web dinamici ad alto volume o sistemi business-critical. Inoltre, molti dei CMS di successo come WordPress e Joomla nascono proprio con il supporto predefinito a MySQL.
Si tratta quindi di un software libero, rilasciato a doppia licenza, GNU GPL e licenza commerciale, ed è sviluppato per essere il più conforme possibile agli standard ANSI SQL; MySQL comprende pertanto un ampio sottoinsieme dell'ANSI SQL 99, così come estensioni, stored procedures, trigger, cursori e viste aggiornabili.
Il programma base è eseguito su un server che fornisce accesso multi-utente a diversi database.
Inoltre è in grado di lavorare sia in sistemi client/server, sia di tipo embedded ed è compatibile con i più noti sistemi operativi, tra i quali Windows, Linux, OS X.

4.4 ORACLE XE (EXPRESS EDITION)

Si tratta dell'edizione gratuita del DBMS di Oracle Corporation.
In quanto tale, offre la maggior parte delle funzionalità previste dalle versioni a pagamento, ma con delle importanti restrizioni, incluse l'utilizzo di un'unica CPU del sistema, l'impiego di massimo 1 GB di RAM, anche se ve n'è una quantità maggiore disponibile, e una limitazione alla memorizzazione dei dati, che possono avere una

[4] Un ORDBMS è un RDBMS basato su un modello di database orientato agli oggetti; supporta pertanto oggetti, classi ed ereditarietà negli schemi e nel linguaggio delle query.
[5] MVCC: metodo di controllo della concorrenza comunemente usato dai sistemi di gestione delle basi di dati per fornire un accesso concorrente ai database.
[6] Point in time recovery (PITR): sistemi in cui un amministratore può ripristinare o recuperare un set di dati da una configurazione ad un determinato momento precedente.
[7] Sito ufficiale: oracle.com.

dimensione massima di 4 GB. Al contempo non supporta viste materializzate, Java virtual machine interna e oggetti partizionati.
Rimangono valide le caratteristiche principali di cui sono dotate le versioni a pagamento; in particolare è presente il supporto di indici, cluster, trigger e stored procedures.
Oracle XE inoltre memorizza i dati logicamente sotto forma di tabelle e fisicamente in file di dati; a livello fisico i dati comprendono uno o più blocchi, la cui dimensione può variare a seconda dei file di dati.
I sistemi operativi supportati sono diversi, tra i quali si citano Windows, Linux, OS X.

5 TEST E VALUTAZIONI

5.1 PIATTAFORMA DI TESTING

I test sono stati condotti su una macchina virtuale dotata di 16 GB di RAM, 550 GB di memoria secondaria, di cui 50 GB di SO e 500 GB di dati, e 8 CPU (Intel Xeon E312xx, Sandy Bridge 2.49GHz).
La macchina virtuale è ospitata sulla piattaforma di calcolo di Cineca denominata PICO [8].
Il sistema operativo è MS Windows Server 2012 R2 Standard a 64 bit.

5.2 BENCHMARKING

L'analisi è condotta su quattro tipologie di informazioni: piani e tempi di esecuzione, numero di thread utilizzati e tempo di CPU impiegato per lo svolgimento di ogni interrogazione sottomessa.

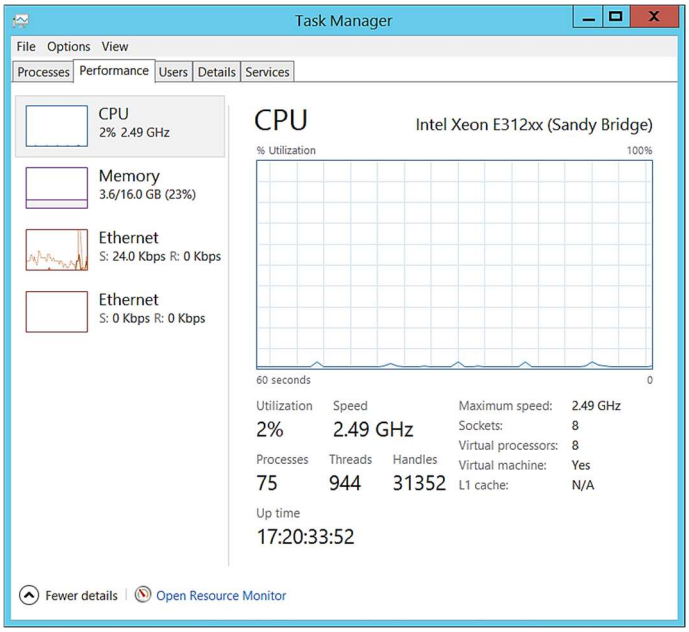


Figura 1

I piani di esecuzione sono ricavati all'interno dei singoli DBMS, eseguendo l'apposito comando da interfaccia grafica o testuale. I tempi di esecuzione sono anch'essi ottenuti direttamente dai DBMS, che per ogni query indicano i secondi trascorsi fino al completamento dell'operazione richiesta. Il numero di thread utilizzati si ricava analizzando i piani di esecuzione.
I dati relativi all'utilizzo della CPU, infine, sono ricavati dal tool MS Windows Task Manager (WTM), con il quale sono equipaggiate tutte le versioni dei sistemi operativi MS Windows.

[8] Nuova infrastruttura, nata nel 2014, che nasce come risposta alla crescente richiesta di servizi e capacità, immagazzinamento dati, gestione, calcolo, visualizzazione, essenziali al fine di affrontare la sfida posta da problemi di tipo "*Big Data*". Maggiori informazioni a questo link.

6 PROCESSO DI TESTING

6.1 STRUTTURA DELLA BASE DI DATI

I dati impiegati nei test comparativi sono stati prelevati da Statistical Computing Statistical Graphics[9].
Si tratta di informazioni tracciate e messe a disposizione dall'organo americano *U.S. Department of Transportation's Bureau of Transportation Statistics*[10] e sono relative ai voli locali registrati nell'arco dell'anno 2008, utilizzate per condurre statistiche sui ritardi dei voli e le loro cause; la natura e il significato di tali informazioni non sono rilevanti ai fini dello studio svolto, ma fungono unicamente da base di dati tramite la quale perseguire e realizzare lo scopo principale dello studio: la valutazione delle performance.

Tali dati, fruibili inizialmente in forma di singola tabella comprendente 7,009,725 righe, sono stati successivamente impiegati per la creazione di otto relazioni, generate operando dapprima una ripartizione delle colonne della tabella originaria, e in seguito applicando opportune condizioni allo scopo di ottenere un numero di righe differenziato per ognuna.
Tali tabelle non sono legate tra loro da specifiche associazioni tra chiavi primarie e chiavi esterne, in quanto lo studio si concentra sulla realizzazione di Full Outer Join con la chiave primaria - campo ID - come chiave di join.
Nell'appendice A sono consultabili le query formulate per la generazione delle tabelle costituenti la base di dati su cui è imperniato lo studio.

Per una visione completa e dettagliata si riporta lo schema delle relazioni impiegate e una tabella riassuntiva contenente le informazioni di maggior rilievo associate ad ognuna di esse.

Nome	*Significato*	*Numero di righe*	*Dimensione*
Flight_1_1	Distanza non specificata o, se lo è, inferiore a 350km	2,006,447	Grande
Flight_1_2	Voli con provenienza o diretti ad "ATL"	829,034	Media/Grande
Flight_1_3	Uniquecarrier='B6'	196,091	Piccola
Flight_2_1	CRSDepTime > 920 e CRSArrTime < 1200	499,799	Media
Flight_2_2	Orario di partenza non disponibile (DepTime='NA')	136,246	Piccola
Flight_2_3	CrsDepTime > 700	6,351,272	Grande
Flight_3_1	Voli cancellati	137,434	Piccola
Flight_3_2	Ritardo > 60	380,530	Piccola/Media

Tabella 1

[9] Sezione di ASA (American Statistical Association). Sito ufficiale.
[10] Per ulteriori informazioni si rimanda al link.

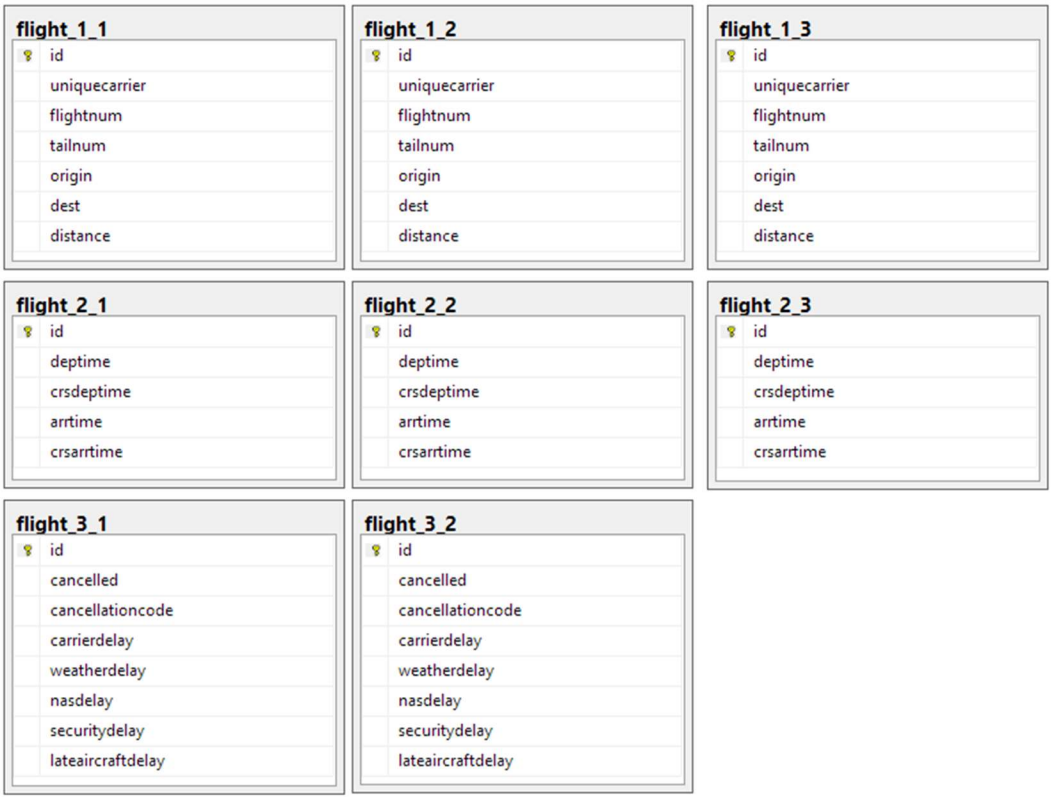


Figura 3

Le tabelle sopra riportate sono state create con una particolare attenzione alla loro dimensione, allo scopo di ottenere un set di relazioni aventi cardinalità differenti; operando in tal modo si è offerta la possibilità di indagare le strategie adottate dai DBMS in base alle dimensioni coinvolte, conducendo, oltre all'analisi comparativa tra i diversi DBMS considerati, un'ulteriore analisi interna al singolo DBMS, che focalizza la propria attenzione sulle variazioni delle strategie impiegate a seconda dei quantitativi di righe da processare ed elaborare.

Una volta creata la base di dati su uno dei DBMS considerati – PostgreSQL –, essa è stata poi migrata sugli altri tre DBMS, in modo da avere un comune database da utilizzare per le interrogazioni successivamente svolte al fine di testarne le performance. Per la migrazione ci si è avvalsi di file di backup, lanciati su SQL Server, MySQL e Oracle XE.
Ove risultato possibile, le tabelle sono state create direttamente, senza effettuare una più lunga migrazione. Nell'appendice B sono riportate le query impiegate a tal proposito.

6.2 METODOLOGIA

Lo studio, come anticipato, è stato svolto secondo una modalità che permettesse di condurre un'analisi in relazione al confronto tra i quattro DBMS presentati e al contempo rendendo possibile un'indagine interna al singolo DBMS, volta all'individuazione e comprensione di variazioni nelle tecniche adottate, in merito a cambiamenti nella cardinalità delle tabelle di volta in volta considerate.
Per concretizzare tale obiettivo si è quindi scelto di suddividere le interrogazioni in quattro macro gruppi, ognuno dei quali considera un set differente di tre tabelle che complessivamente

determinano un numero globale di righe da processare diverso per ogni gruppo, che permette di coprire in maniera sufficientemente omogenea uno spettro di indicativamente 9 milioni di righe.

Gruppo	*Tabelle coinvolte*	*Numero massimo di righe elaborate*
Gruppo 1	1. flight_2_3: **6,351,272** righe 2. flight_1_1: **2,006,447** righe 3. flight_1_2: **829,034** righe	9,186,753
Gruppo 2	1. flight_1_1: **2,006,447** righe 2. flight_1_2: **829,034** righe 3. flight_2_1: **499,799** righe	3,335,280
Gruppo 3	1. flight_3_2: **380,530** righe 2. flight_1_2: **829,034** righe 3. flight_2_1: **499,799** righe	1,709,363
Gruppo 4	1. flight_2_1: **499,799** righe 2. flight_1_3: **196,091** righe 3. flight_3_1, **137,434** righe	833,324

Tabella 2

Lo studio si suddivide in due parti distinte: la prima, su cui si concentra la porzione maggiore dello studio stesso, è volta al confronto delle performance in assenza di indici appositi, la seconda invece aggiunge alle precedenti osservazioni ulteriori considerazioni, derivate dall'impiego di indici appositamente creati per incrementare le prestazioni dei DBMS, in modo da verificare e rapportare gli aumenti in termini di prestazioni, registrati grazie alla presenza degli indici, e determinare quanto gli indici stessi possano impattare positivamente sui tempi di esecuzione.

Relativamente alla prima parte, per ogni gruppo sono state immesse diverse interrogazioni, con due operazioni di Full Outer Join in sequenza come impalcatura comune ad ognuna: inizialmente senza specificare alcuna condizione nella clausola WHERE, in modo da testare i comportamenti adottati in caso di Full Outer Join puri, e successivamente riproponendo le medesime query ma imponendo delle condizioni, restrittive o meno, che potessero sia trasformare i Full Outer Join in Right/Left Join, in modo da analizzare le scelte operative attuate dai DBMS in merito a tali casistiche, sia condurre ad una riduzione del numero di righe da elaborare, al fine di fornire una maggiore copertura al range di dimensioni entro il quale sono testati i comportamenti dei singoli DBMS.
Le interrogazioni sono state immesse parallelamente nei quattro DBMS; per ogni query sono state poi registrate le informazioni relative all'utilizzo di CPU, al numero di threads impiegati e ai tempi e i relativi piani di esecuzione prodotti da ognuno dei DBMS. Le prime tre tipologie di dati ricavati sono stati impiegati per il calcolo di statistiche e la produzione di grafici che potessero rendere in forma numerica e visiva le differenze prestazionali dei DBMS coinvolti nello studio, mentre i piani di esecuzione sono stati analizzati in maniera dettagliata al fine di comprendere in maniera più approfondita la natura delle discrepanze prestazionali registrate, per unire ad una semplice osservazione dei primi dati menzionati, anche un esame delle motivazioni che hanno portato a tali dati.

Per quanto riguarda la seconda parte, anche in questo caso si mantiene una distinzione basata sulle dimensioni delle tabelle coinvolte, in modo da registrare eventuali differenze nelle strategie adottate a seconda della cardinalità considerata. Gli indici sono stati creati su colonne interessate da una condizione di filtro, vagliando diversi casi a seconda della natura più o meno restrittiva di tale filtro. Si sono registrati i dati relativi ai tempi di esecuzione, utilizzati in seguito sia per determinare quanto gli indici comportino un incremento delle performance all'interno dello stesso DBMS, sia per realizzare un'indagine comparativa più completa tra diversi DBMS.

Nella presente tesi sono riportati solo i piani di esecuzione e le informazioni maggiormente significative, per una migliore leggibilità del documento, che si pone lo scopo di rendere fruibili i dati registrati senza dilungarsi sull'analisi dettagliata delle tecniche adottate dai singoli DBMS. Tuttavia la trattazione più completa si può consultare.

7 ANALISI COMPARATIVA

Le successive pagine sono strutturate nel seguente modo: viene innanzitutto riportata la query eseguita, in una doppia versione, di cui una compatibile con la versione del linguaggio SQL supportata da MySQL – che non prevede Full Outer Join.
L'interrogazione è correlata dai dati principali ricavati contestualmente per ogni DBMS. I dati comprendono tempi di esecuzione registrati e query plan. Questi ultimi sono riportati solamente nei casi di maggiore interesse, ovvero qualora presentino degli aspetti degni di nota, mentre piani di esecuzione analoghi ad altri già precedentemente inseriti sono tralasciati, in modo da rendere più organica la lettura dei punti principali su cui sono effettivamente basate le considerazioni. Per le stesse motivazioni, i query plan non sono analizzati e indagati dettagliatamente, ma ne viene eventualmente riportata una breve sintesi.
Successivamente sono presentate le osservazioni principali, in un contesto che pone in comparazione le strategie e gli aspetti di maggior rilievo, individuati nella precedente analisi, adottati dai quattro DBMS oggetto di studio.
Si ricorda ulteriormente che l'analisi completa è consultabile.

7.1 INTERROGAZIONI SENZA APPOSITI INDICI

7.1.1 Gruppo 1

7.1.1.1 Query 1 – nessuna condizione imposta

La prima query è stata impostata secondo due diverse formulazioni: la prima non prevede l'utilizzo di COALESCE, mentre la seconda sì. Soltanto MS SQL Server e Oracle XE supportano la versione senza COALESCE.

```
SELECT *
FROM flight_2_3 f23
FULL OUTER JOIN flight_1_1 f11 ON f11.id=f23.id
FULL OUTER JOIN flight_1_2 f12 ON (f12.id=f11.id OR f12.id=f23.id)
```

- *Righe ritornate:* ***6,599,143***
- *Tempi di esecuzione:*

MS SQL Server	*PostgreSQL*	*Oracle*	*MySQL*
00:02:55	-	*00:43:49*	-

- *Piani di esecuzione*
  MS SQL Server:

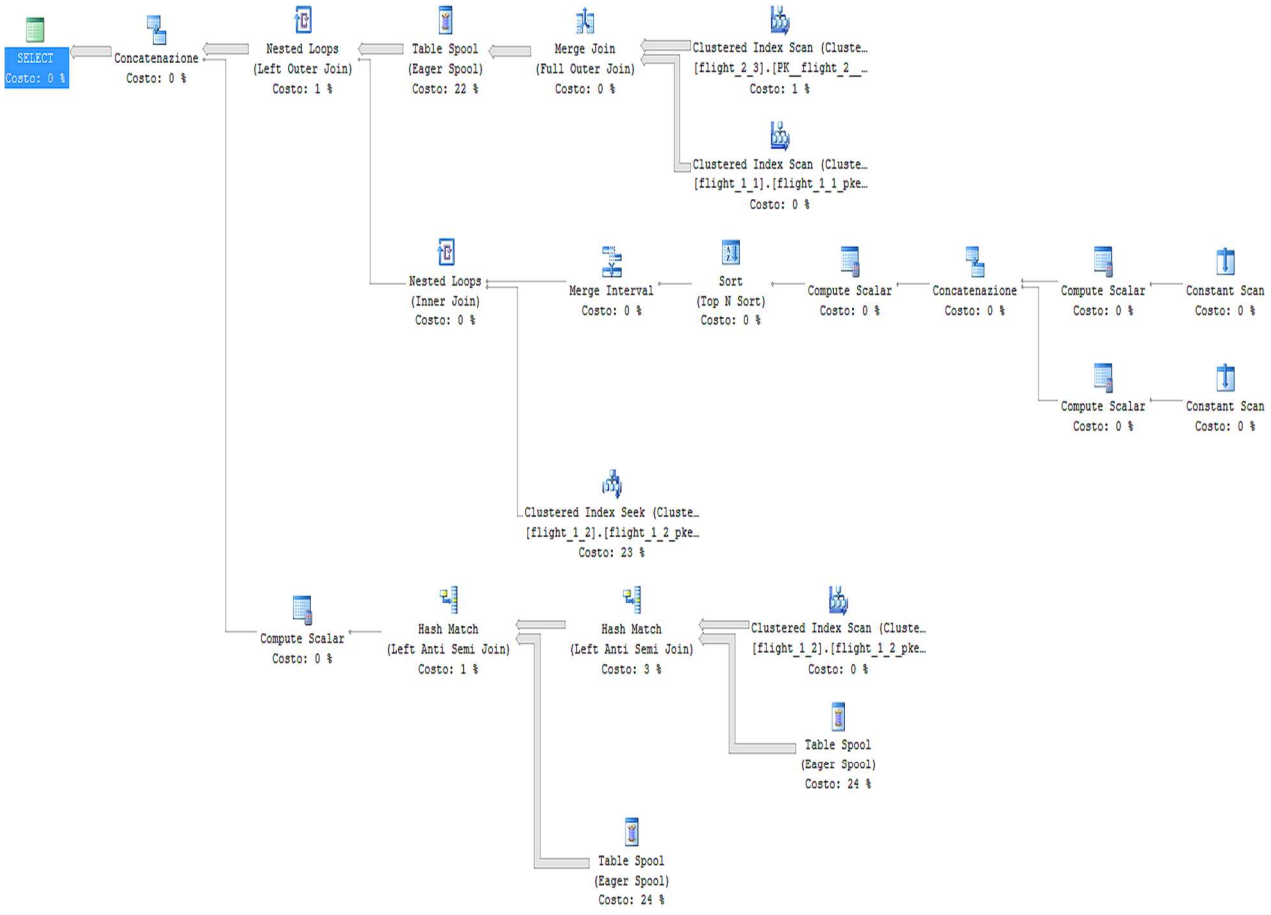


Figura 4

Oracle XE:

Operation	Object	Optimizer	Cost	Cardinality	Bytes
◢ SELECT STATEMENT		ALL_ROWS	38,277,421	13,531,575	6,941,697,975
◢ VIEW			38,277,421	13,531,575	6,941,697,975
◢ UNION-ALL			0	0	0
◢ NESTED LOOPS (OUTER)			38,160,143	12,702,542	6,668,834,550
◢ VIEW	VW_FOJ_0		44,501	6,351,271	2,134,027,056
◢ HASH JOIN (FULL OUTER)			44,501	6,351,271	711,342,352
TABLE ACCESS (FULL)	FLIGHT_1_1	ANALYZED	4,839	2,006,446	120,386,760
TABLE ACCESS (FULL)	FLIGHT_2_3	ANALYZED	13,575	6,351,271	330,266,092
◢ VIEW			6	2	378
◢ CONCATENATION			0	0	0
◢ TABLE ACCESS (BY INDEX ROWID)	FLIGHT_1_2	ANALYZED	3	1	59
INDEX (UNIQUE SCAN)	FLIGHT_1_2_PKEY	ANALYZED	2	1	0
◢ TABLE ACCESS (BY INDEX ROWID)	FLIGHT_1_2	ANALYZED	3	1	59
INDEX (UNIQUE SCAN)	FLIGHT_1_2_PKEY	ANALYZED	2	1	0
◢ HASH JOIN (ANTI)			117,278	829,033	71,296,838
TABLE ACCESS (FULL)	FLIGHT_1_2	ANALYZED	2,002	829,034	48,913,006
◢ VIEW	VW_SQ_1		89,002	12,702,542	342,968,634
◢ UNION-ALL			0	0	0
◢ VIEW	VW_FOJ_1		44,501	6,351,271	171,484,317
◢ HASH JOIN (FULL OUTER)			44,501	6,351,271	711,342,352
TABLE ACCESS (FULL)	FLIGHT_1_1	ANALYZED	4,839	2,006,446	120,386,760
TABLE ACCESS (FULL)	FLIGHT_2_3	ANALYZED	13,575	6,351,271	330,266,092
◢ VIEW	VW_FOJ_1		44,501	6,351,271	171,484,317
◢ HASH JOIN (FULL OUTER)			44,501	6,351,271	711,342,352
TABLE ACCESS (FULL)	FLIGHT_1_1	ANALYZED	4,839	2,006,446	120,386,760
TABLE ACCESS (FULL)	FLIGHT_2_3	ANALYZED	13,575	6,351,271	330,266,092

Figura 5

Con l'introduzione di COALESCE si possono registrare aumenti prestazionali.

```
SELECT *
FROM flight_2_3 f23
FULL OUTER JOIN flight_1_1 f11 ON f11.id=f23.id
FULL OUTER JOIN flight_1_2 f12 ON f12.id=COALESCE(f11.id, f23.id)
```

Versione per MySQL:

```
SELECT f23. * , f11. * , f12. *
FROM flight_2_3 f23
LEFT OUTER JOIN flight_1_1 f11 USING (id)
LEFT OUTER JOIN flight_1_2 f12 USING (id)
UNION
SELECT f23. * , f11. * , f12. *
FROM flight_1_1 f11
LEFT OUTER JOIN flight_2_3 f23 USING (id)
LEFT OUTER JOIN flight_1_2 f12 USING (id)
UNION
SELECT f23. * , f11. * , f12. *
FROM flight_1_2 f12
LEFT OUTER JOIN flight_2_3 f23 USING (id)
LEFT OUTER JOIN flight_1_1 f11 USING (id)
```

- *Righe ritornate:* ***6,599,143***
- *Tempi di esecuzione:*

MS SQL Server	*PostgreSQL*	*Oracle*	*MySQL*
00:01:18	*00:11:37*	*00:37:39*	*26:13:00* *00:03:58 (vista intermedia)*

- *Piani di esecuzione*
  MS SQL Server:

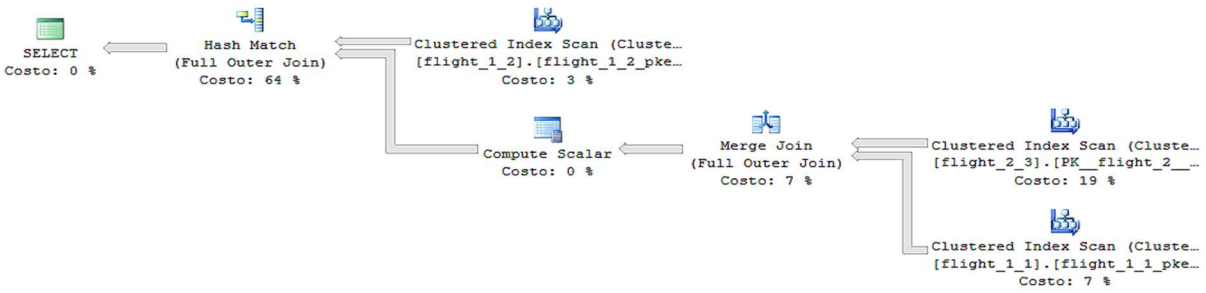


Figura 6

L'introduzione di COALESCE permette di ridurre la complessità del piano prodotto, evitando l'impiego di un'operazione di Index Seek dinamico.
Il tempo di esecuzione diminuisce del 53.72% rispetto il precedente.

PostgreSQL

1	QUERY PLAN
2	Hash Full Join (cost=124314.32..730858.47 rows=6351328 width=161)
3	Hash Cond: (COALESCE(f11.id, f23.id) = f12.id)
4	-> Hash Full Join (cost=87375.06..399202.45 rows=6351328 width=104)
5	Hash Cond: (f23.id = f11.id)
6	-> Seq Scan on flight_2_3 f23 (cost=0.00..128765.28 rows=6351328 width=48)
7	-> Hash (cost=42699.47..42699.47 rows=2006447 width=56)
8	-> Seq Scan on flight_1_1 f11 (cost=0.00..42699.47 rows=2006447 width=56)
9	-> Hash (cost=17670.34..17670.34 rows=829034 width=57)
10	-> Seq Scan on flight_1_2 f12 (cost=0.00..17670.34 rows=829034 width=57)

Figura 7

Oracle XE

Operation	Object	Optimizer	Cost	Cardinality	Bytes
◢ SELECT STATEMENT		ALL_ROWS	150,214	13,531,576	6,941,698,488
◢ VIEW	VW_FOJ_0		150,214	13,531,576	6,941,698,488
◢ HASH JOIN (FULL OUTER)			150,214	13,531,576	5,182,593,608
TABLE ACCESS (FULL)	FLIGHT_1_2	ANALYZED	2,002	829,034	48,913,006
◢ VIEW	VW_FOJ_1		44,501	6,351,271	2,057,811,804
◢ HASH JOIN (FULL OUTER)			44,501	6,351,271	711,342,352

Figura 8

Anche per Oracle XE si registra una diminuzione del tempo di esecuzione pari al 14.07%, che comporta pertanto un aumento delle prestazioni associate.

MySQL

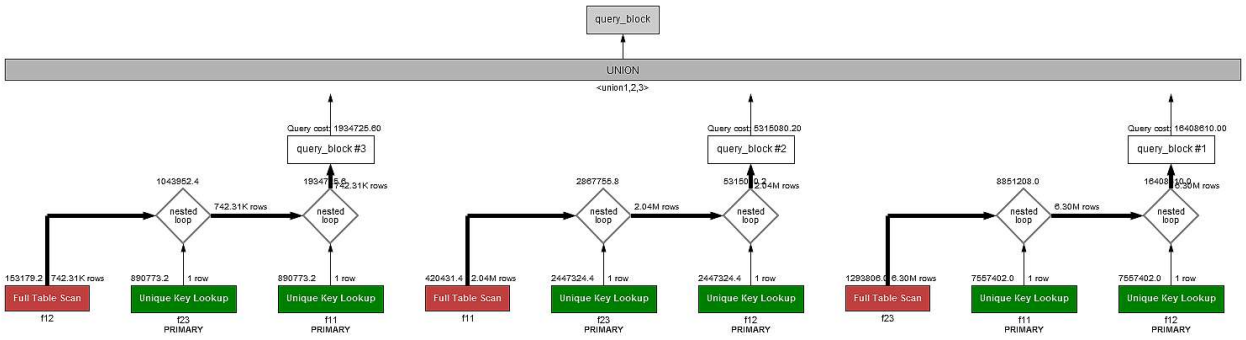


Figura 9

Per MySQL si possono ottenere gli stessi risultati ma con prestazioni migliori, sfruttando delle **viste intermedie**:

```
Create view resPartialJoin as
SELECT f12.*
FROM flight_1_2 f12
LEFT OUTER JOIN flight_1_1 f11 ON f12.id=f11.id
UNION
SELECT f12.*
FROM flight_1_1 f11
LEFT OUTER JOIN flight_1_2 f12 ON f12.id=f11.id;

SELECT v1.* , f23.*
FROM resPartialJoin v1
LEFT OUTER JOIN flight_2_3  f23 on v1.id=f23.id
UNION
SELECT v1.* , f23.*
FROM flight_2_3 f23
LEFT OUTER JOIN resPartialJoin v1 on v1.id=f23.id;
```

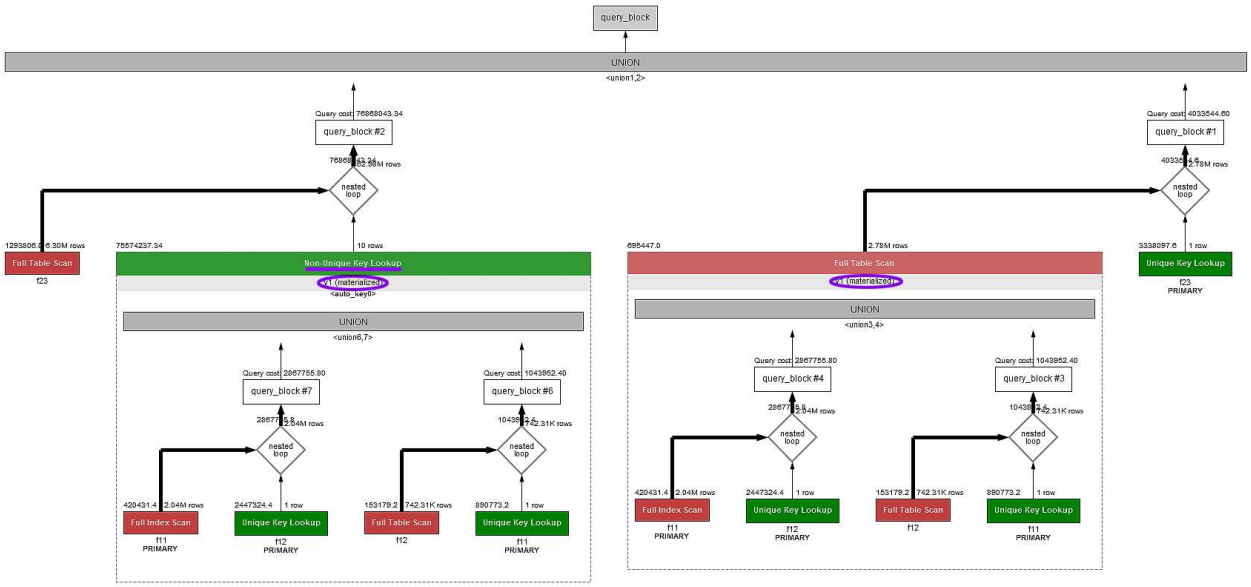


Figura 10

Il piano di esecuzione risulta più complesso, ma al tempo stesso si registra una drastica riduzione dei tempi di esecuzione, pari al 99.7%.

Confrontando i costi complessivi associati alle query di Figura 9 e Figura 10, si nota che il costo maggiore è quello relativo alla query che sfrutta la vista intermedia, infatti quest'ultimo è pari a 80,901,587, contro il costo della formulazione precedente pari a 23,658,415.

Rispetto la versione precedente, quella attuale però registra delle prestazioni migliori, con un tempo di esecuzione inferiore; ciò è determinato dal fatto che in realtà nel secondo blocco di Figura 10, la vista v1 non è nuovamente determinata, ma viene sfruttata la tabella temporanea creata precedentemente in merito al primo blocco. Infatti, v1 è una vista *materialized*, ovvero il suo contenuto, la prima volta che viene determinato, è memorizzato in una tabella temporanea; in questo modo, quando MySQL avrà bisogno successivamente del risultato, potrà semplicemente fare riferimento alla tabella temporanea, velocizzando e rendendo maggiormente efficiente l'esecuzione della query e quindi apportando benefici relativamente alle prestazioni.

7.1.1.2 Query 2 – singola condizione poco restrittiva

```
SELECT *
FROM flight_2_3 f23
FULL OUTER JOIN flight_1_1 f11 ON f11.id=f23.id
FULL OUTER JOIN flight_1_2 f12 ON (f12.id=f11.id OR f12.id=f23.id)
WHERE f23.deptime>'1500'
```

Versione per MySQL:

```
SELECT f23. * , f11. * , f12. *
FROM flight_2_3 f23
LEFT OUTER JOIN flight_1_1 f11 on f23.id=f11.id
LEFT OUTER JOIN flight_1_2 f12 on f23.id=f12.id
WHERE f23.deptime>'1500'
UNION
SELECT f23. * , f11. * , f12. *
FROM flight_1_1 f11
LEFT OUTER JOIN flight_2_3 f23 on f23.id=f11.id
LEFT OUTER JOIN flight_1_2 f12 on f12.id=f11.id
WHERE f23.deptime>'1500'
UNION
SELECT f23. * , f11. * , f12. *
FROM flight_1_2 f12
LEFT OUTER JOIN flight_2_3 f23 on f23.id=f12.id
LEFT OUTER JOIN flight_1_1 f11 on f12.id=f11.id
WHERE f23.deptime>'1500'
```

- *Righe ritornate:* **4,184,487**
- *Tempi di esecuzione:*

MS SQL Server	*PostgreSQL*	*Oracle*	*MySQL*
00:00:49	*00:07:29*	*00:45:42*	*00:38:15*

- *Piani di esecuzione*

  MS SQL Server:

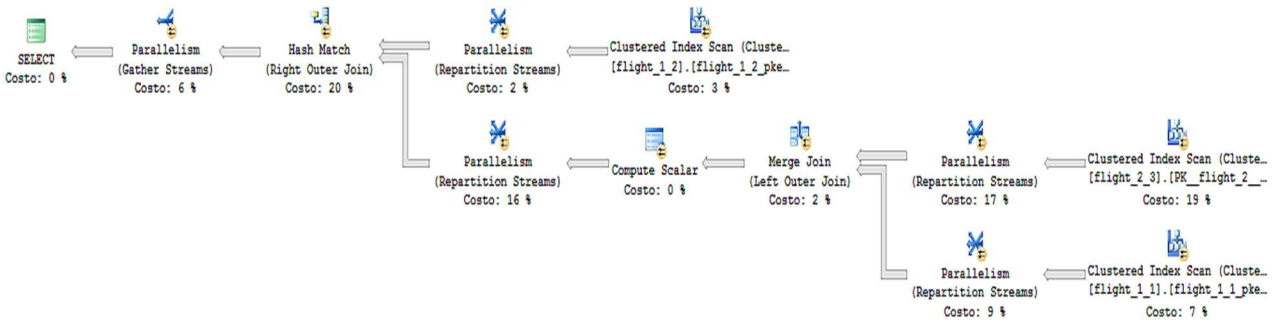


Figura 11

PostgreSQL

1	QUERY PLAN
2	Hash Left Join (cost=124314.32..594324.82 rows=4196699 width=161)
3	Hash Cond: (COALESCE(f11.id, f23.id) = f12.id)
4	-> Hash Left Join (cost=87375.06..359626.94 rows=4196699 width=104)
5	Hash Cond: (f23.id = f11.id)
6	-> Seq Scan on flight_2_3 f23 (cost=0.00..144643.60 rows=4196699 width=48)
7	Filter: ((deptime)::text > '1500'::text)
8	-> Hash (cost=42699.47..42699.47 rows=2006447 width=56)
9	-> Seq Scan on flight_1_1 f11 (cost=0.00..42699.47 rows=2006447 width=56)
10	-> Hash (cost=17670.34..17670.34 rows=829034 width=57)
11	-> Seq Scan on flight_1_2 f12 (cost=0.00..17670.34 rows=829034 width=57)

Figura 12

Oracle XE

Operation	Object	Optimizer	Cost	Cardinality	Bytes
SELECT STATEMENT		ALL_ROWS	149,349	6,306,163	2,415,260,429
HASH JOIN (RIGHT OUTER)			149,349	6,306,163	2,415,260,429
TABLE ACCESS (FULL)	FLIGHT_1_2	ANALYZED	2,002	829,034	48,913,006
VIEW	VW_FOJ_0		44,372	6,306,163	2,043,196,812
HASH JOIN (RIGHT OUTER)			44,372	6,306,163	706,290,256
TABLE ACCESS (FULL)	FLIGHT_1_1	ANALYZED	4,839	2,006,446	120,386,760
TABLE ACCESS (FULL)	FLIGHT_2_3	ANALYZED	13,584	6,306,163	327,920,476

Figura 13

MySQL

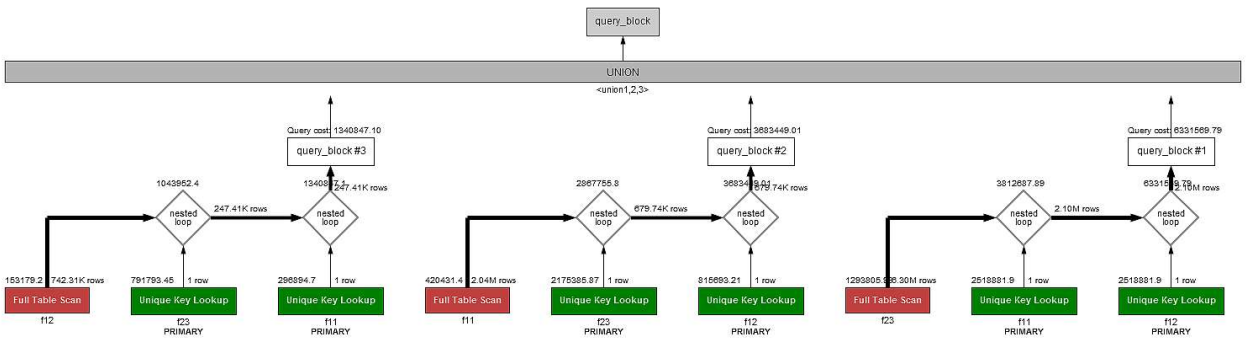


Figura 14

7.1.1.3 Query 3 – 2 condizioni poco restrittive

```
SELECT *
FROM flight_2_3 f23
FULL OUTER JOIN flight_1_1 f11 ON f11.id=f23.id
FULL OUTER JOIN flight_1_2 f12 ON f12.id=COALESCE(f11.id, f23.id)
WHERE f23.deptime>'1500'
AND f12.flightnum>'1000'
```

Versione per MySQL:

```
SELECT f23. * , f11. * , f12. *
FROM flight_2_3 f23
LEFT OUTER JOIN flight_1_1 f11 on f23.id=f11.id
LEFT OUTER JOIN flight_1_2 f12 on f23.id=f12.id
WHERE f23.deptime>'1500'
AND f12.flightnum>'1000'
UNION
SELECT f23. * , f11. * , f12. *
FROM flight_1_1 f11
LEFT OUTER JOIN flight_2_3 f23 on f23.id=f11.id
LEFT OUTER JOIN flight_1_2 f12 on f12.id=f11.id
WHERE f23.deptime>'1500'
AND f12.flightnum>'1000'
```

- *Righe ritornate:* **481,357**
- *Tempi di esecuzione:*

MS SQL Server	*PostgreSQL*	*Oracle*	*MySQL*
00:00:06	*00:01:32*	*00:02:01*	*00:01:28*

- *Piani di esecuzione*

  MS SQL Server

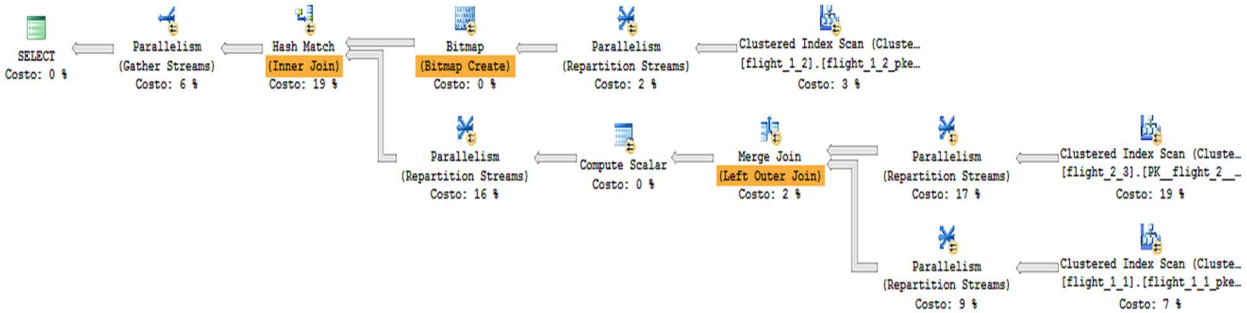


Figura 15

PostgreSQL

```
1  QUERY PLAN
2  Hash Join (cost=126220.63..595792.59 rows=4160545 width=161)
3    Hash Cond: (COALESCE(f11.id, f23.id) = f12.id)
4    -> Hash Left Join  (cost=87375.06..359626.94 rows=4196699 width=104)
5         Hash Cond: (f23.id = f11.id)
6         -> Seq Scan on flight_2_3 f23  (cost=0.00..144643.60 rows=4196699 width=48)
7              Filter: ((deptime)::text > '1500'::text)
8         -> Hash  (cost=42699.47..42699.47 rows=2006447 width=56)
9              -> Seq Scan on flight_1_1 f11  (cost=0.00..42699.47 rows=2006447 width=56)
10   -> Hash  (cost=19742.93..19742.93 rows=821892 width=57)
11        -> Seq Scan on flight_1_2 f12  (cost=0.00..19742.93 rows=821892 width=57)
12             Filter: ((flightnum)::text > '1000'::text)
```

Figura 16

Oracle XE

Operation	Object	Optimizer	Cost	Cardinality	Bytes
◢ SELECT STATEMENT		ALL_ROWS	149,350	6,306,163	2,415,260,429
◢ HASH JOIN			149,350	6,306,163	2,415,260,429
TABLE ACCESS (FULL)	FLIGHT_1_2	ANALYZED	2,003	829,034	48,913,006
◢ VIEW	VW_FOJ_0		44,372	6,306,163	2,043,196,812
◢ HASH JOIN (RIGHT OUTER)			44,372	6,306,163	706,290,256
TABLE ACCESS (FULL)	FLIGHT_1_1	ANALYZED	4,839	2,006,446	120,386,760
TABLE ACCESS (FULL)	FLIGHT_2_3	ANALYZED	13,584	6,306,163	327,920,476

Figura 17

MySQL

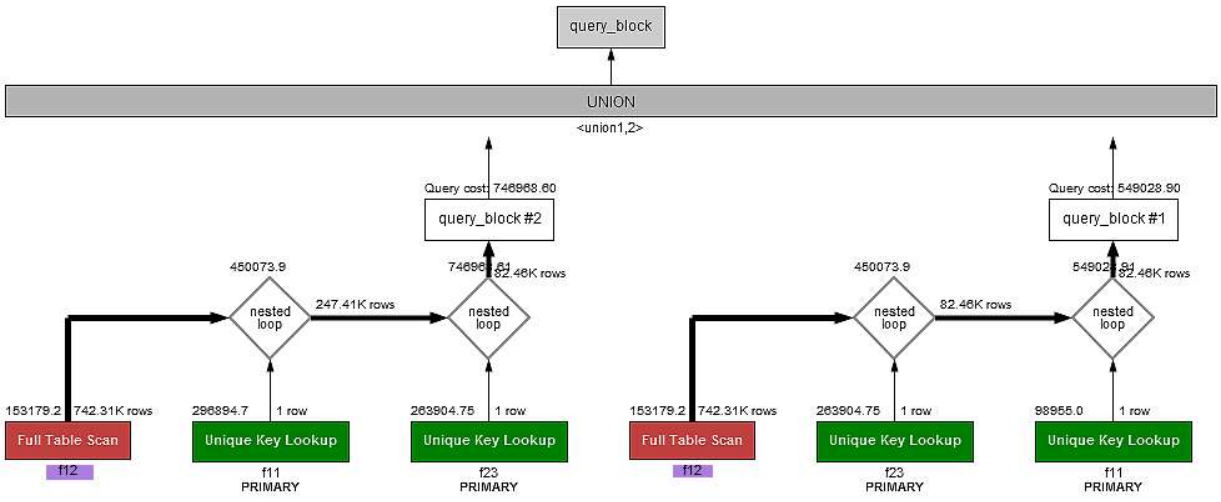


Figura 18

7.1.1.4 Query 4 – 3 condizioni, di cui una restrittiva

```
SELECT *
FROM flight_2_3 f23
FULL OUTER JOIN flight_1_1 f11 ON f11.id=f23.id
FULL OUTER JOIN flight_1_2 f12 ON f12.id=COALESCE(f11.id, f23.id)
WHERE f23.deptime>'1500'
AND f11.uniquecarrier='EV' OR f11.uniquecarrier='DL'
AND f12.flightnum>'1000'
```

Versione per MySQL:

```
SELECT *
FROM flight_1_1 f11
LEFT OUTER JOIN flight_2_3 f23 on f23.id=f11.id
LEFT OUTER JOIN flight_1_2 f12 on f12.id=COALESCE(f11.id, f23.id)
WHERE f23.deptime>'1500'
AND f11.uniquecarrier='EV' OR f11.uniquecarrier='DL'
AND f12.flightnum>'1000'
```

- *Righe ritornate:* **110,753**
- *Tempi di esecuzione:*

MS SQL Server	*PostgreSQL*	*Oracle*	*MySQL*
00:00:03	*00:00:34.5*	*00:01:22*	*00:00:12*

- *Piani di esecuzione*

  MS SQL Server

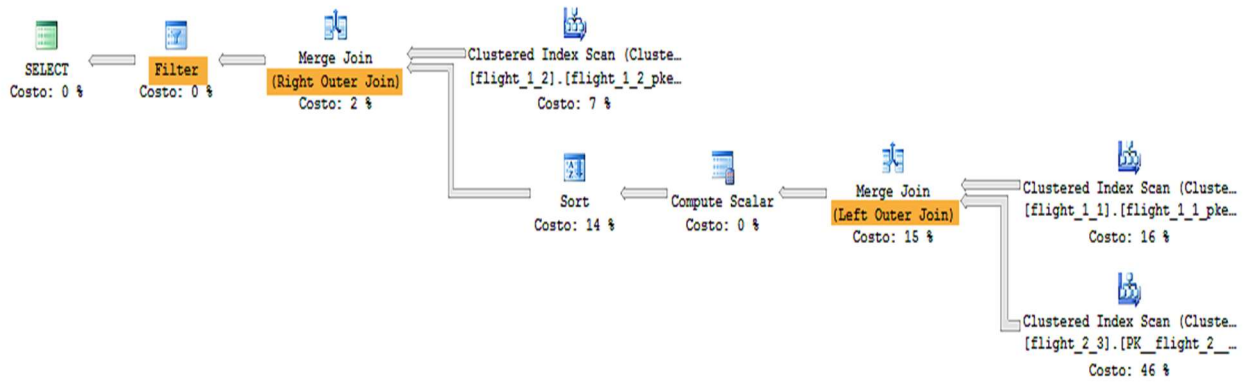


Figura 19

PostgreSQL

```
1  QUERY PLAN
2  Hash Left Join (cost=93560.01..379752.97 rows=12194 width=161)
3    Hash Cond: (COALESCE(f11.id, f23.id) = f12.id)
4    Filter: ((((f23.deptime)::text > '1500'::text)
     AND ((f11.uniquecarrier)::text = 'EV'::text)) OR (((f11.uniquecarrier)::text = 'DL'::text)
     AND ((f12.flightnum)::text > '1000'::text)))
5    -> Hash Right Join  (cost=56620.74..324301.93 rows=174643 width=104)
6        Hash Cond: (f23.id = f11.id)
7        -> Seq Scan on flight_2_3 f23  (cost=0.00..128765.28 rows=6351328 width=48)
8        -> Hash  (cost=52731.71..52731.71 rows=174643 width=56)
9            -> Seq Scan on flight_1_1 f11  (cost=0.00..52731.71 rows=174643 width=56)
10               Filter: (((uniquecarrier)::text = 'EV'::text) OR ((uniquecarrier)::text = 'DL'::text))
11 -> Hash  (cost=17670.34..17670.34 rows=829034 width=57)
12       -> Seq Scan on flight_1_2 f12  (cost=0.00..17670.34 rows=829034 width=57)
```

Figura 20

Oracle XE

```
| 0 | SELECT STATEMENT      |          |     |     |  150K(100)|      |     |     |      |
|* 1 | VIEW                 | VW_FOJ_0 |  13M| 6620M|     |  150K  (1)| 00:30:03 |     |     |     |     |
|* 2 |  HASH JOIN FULL OUTER |          |  13M| 4942M|  56M|  150K  (1)| 00:30:03 |  75M| 8625K|  68M (1)|  125K|
| 3 |   TABLE ACCESS FULL   | FLIGHT_1_2 |  829K|  46M|     | 2002   (1)| 00:00:25 |     |     |     |     |
| 4 |  VIEW                 | VW_FOJ_1 |  6351K| 1962M|     | 44501  (1)| 00:08:55 |     |     |     |     |
|* 5 |   HASH JOIN FULL OUTER|          |  6351K|  678M|  137M| 44501  (1)| 00:08:55 |  176M| 8625K|  78M (1)|  253K|
| 6 |    TABLE ACCESS FULL  | FLIGHT_1_1 |  2006K|  114M|     | 4839   (1)| 00:00:59 |     |     |     |     |
| 7 |    TABLE ACCESS FULL  | FLIGHT_2_3 |  6351K|  314M|     | 13575  (1)| 00:02:43 |     |     |     |     |
---------------------------------------------------------------------------------------------------------------------------

  1 - filter((("F23"."DEPTIME">'1500' AND "F11"."UNIQUECARRIER"='EV') OR ("F11"."UNIQUECARRIER"='DL' AND
        "F12"."FLIGHTNUM">'1000')))
  2 - access("F12"."ID"=COALESCE("F11"."ID","F23"."ID"))
  5 - access("F11"."ID"="F23"."ID")
```

Figura 21

MySQL

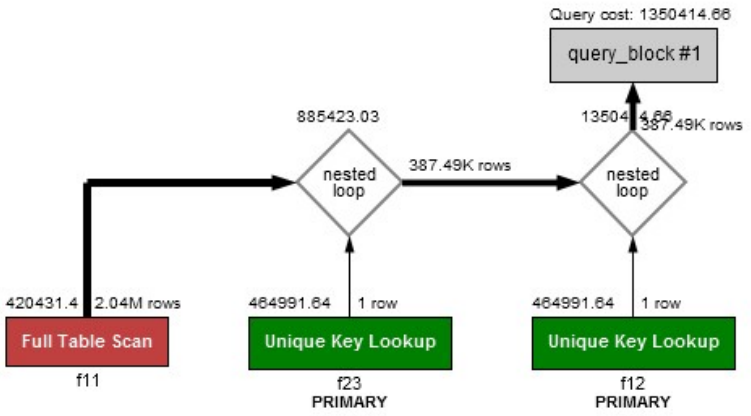


Figura 22

Dato il divario consistente tra MS SQL Server, PostgreSQL e Oracle XE da una parte e MySQL dall'altra, le osservazioni sono condotte considerando dapprima i primi tre DBMS citati, e successivamente integrate con le considerazioni relative a MySQL.

7.1.1.5 Comparazione tra MS SQL Server, PostgreSQL e Oracle XE

Innanzitutto, una prima grande differenza che si può riscontrare è il fatto che sia Oracle XE, sia SQL Server supportano due tipi di sintassi per esprimere il Full Outer Join tra più di due relazioni, infatti può essere espresso sfruttando o meno COALESCE. In linea generale la versione che non prevede COALESCE risulta meno performante, tant'è che la stessa query viene eseguita da SQL Server in questo modo in 2 minuti e 55 secondi, mentre nella versione con COALESCE viene eseguita in 1 minuto e 18 secondi; mentre Oracle XE esegue la versione con COALESCE in 37 minuti e 39 secondi, e in 43 minuti e 49 secondi quella senza.
Senza COALESCE le prestazioni risultano quindi peggiori, pertanto potrebbe essere ritenuta inutile questa versione; il fatto che però sia supportata offre comunque una strada alternativa e dimostra alcuni aspetti della potenza di SQL Server e Oracle XE.
Per quanto riguarda SQL Server infatti, il piano di esecuzione relativo alla query che non sfrutta COALESCE (Figura 4) mostra alcune funzionalità molto interessanti e rilevanti nell'ambito dell'analisi delle prestazioni:

1. **Table Spool**: realizza uno storage temporaneo dei dati per un loro successivo riuso all'interno del query plan; in particolare SQL Server genera uno spool quando il Query Optimizer pensa di poter lavorare più efficientemente con un subset di dati semipermanenti, piuttosto che eseguire multiple operazioni di seek o scan sulle tabelle interessate.
   Si tratta di un'operazione quindi che può influire positivamente in termini di prestazioni, in quanto consente di evitare un'ulteriore lettura completa degli stessi dati, a favore di un loro riuso.
2. **Index Seek dinamico**: si tratta di un'operazione non ancora effettivamente implementata ma resa comunque possibile tramite un *workaround*; in questo modo è realizzabile un Index Seek[11] anche nel caso in cui i valori del range di copertura per la ricerca siano determinabili solo a tempo di esecuzione.
   Questo aspetto tra l'altro mostra anche che SQL Server tende a creare dei piani di esecuzione poco probabili e anche piuttosto complessi e articolati al fine di far prevalere l'operatore di Index Seek su quello di Index Scan[12], meno performante, qualora sia possibile. Ovviamente si tratta di un aspetto positivo nell'ambito delle prestazioni, in quanto permette di ridurre il numero di accessi al disco; deve, tuttavia, essere accompagnato da statistiche aggiornate e corrette, specialmente in relazione alla cardinalità delle tabelle, perché l'operazione di Index Seek può risultare più onerosa rispetto a Index Scan qualora le stime utilizzate siano basate su informazioni non corrette o non aggiornate.
   Si riportano a tal proposito i dati ottenuti tramite l'esecuzione di operazioni di Index Seek

[11] Index Seek: ritorna una selezione di righe della tabella interessata.
[12] Index Scan: ritorna tutte le righe della tabella.

in presenza di statistiche errate, come accaduto in Figura 4, rapportati a quelli ottenuti forzando la realizzazione di Index Scan:

flight_1_2

Clustered Index Scan (Clustered) Esegue l'analisi di un indice cluster, considerando l'intero indice o un intervallo di righe.		Clustered Index Seek (Clustered) Esegue l'analisi di un intervallo di righe specifico in un indice cluster.	
Operazione fisica	Clustered Index Scan	**Operazione fisica**	Clustered Index Seek
Operazione logica	Clustered Index Scan	**Operazione logica**	Clustered Index Seek
Modalità di esecuzione effettiva	Row	**Modalità di esecuzione effettiva**	Row
Numero effettivo di righe	829034	**Numero effettivo di righe**	765164
Numero effettivo di batch	0	**Numero effettivo di batch**	0
Costo I/O stimato	5,88387	**Costo I/O stimato**	0,003125
Costo stimato operatore	6,79596 (0%)	**Costo stimato operatore**	1030,58 (23%)
Costo CPU stimato	0,912094	**Costo CPU stimato**	0,0001581
Costo stimato sottoalbero	6,79596	**Costo stimato sottoalbero**	1030,58
Numero stimato di esecuzioni	1	**Numero stimato di esecuzioni**	6361611
Numero di esecuzioni	1	**Numero di esecuzioni**	6535272
Numero stimato di righe	829034	**Numero stimato di righe**	1
Dimensioni stimate righe	210 B	**Dimensioni stimate righe**	204 B
Riassociazioni effettive	0	**Riassociazioni effettive**	0
Ripristini effettivi	0	**Ripristini effettivi**	0
Ordinato	True	**Ordinato**	True
ID nodo	4	**ID nodo**	37

Figura 23

flight_2_3

Clustered Index Scan (Clustered) Esegue l'analisi di un indice cluster, considerando l'intero indice o un intervallo di righe.		Clustered Index Seek (Clustered) Esegue l'analisi di un intervallo di righe specifico in un indice cluster.	
Operazione fisica	Clustered Index Scan	**Operazione fisica**	Clustered Index Seek
Operazione logica	Clustered Index Scan	**Operazione logica**	Clustered Index Seek
Modalità di esecuzione effettiva	Row	**Modalità di esecuzione effettiva**	Row
Numero effettivo di righe	6351271	**Numero effettivo di righe**	2401092
Numero effettivo di batch	0	**Numero effettivo di batch**	0
Costo I/O stimato	39,5239	**Costo I/O stimato**	0,003125
Costo stimato operatore	46,5104 (1%)	**Costo stimato operatore**	489,078 (28%)
Costo CPU stimato	6,98656	**Costo CPU stimato**	0,0001581
Costo stimato sottoalbero	46,5104	**Costo stimato sottoalbero**	489,078
Numero stimato di esecuzioni	1	**Numero stimato di esecuzioni**	2038881
Numero di esecuzioni	1	**Numero di esecuzioni**	2648963
Numero stimato di righe	6351270	**Numero stimato di righe**	1
Dimensioni stimate righe	155 B	**Dimensioni stimate righe**	150 B
Riassociazioni effettive	0	**Riassociazioni effettive**	0
Ripristini effettivi	0	**Ripristini effettivi**	0
Ordinato	True	**Ordinato**	True
ID nodo	4	**ID nodo**	37

Figura 24

In entrambi i casi, quindi, l'operazione di Index Seek risulta più costosa di Index Scan. La scelta di optare per Index Seek dinamico è infatti dettata dal fatto che il query optimizer stima che il numero di righe ritornate dall'operazione sia solamente 1, mentre in realtà si può notare che per flight_1_2 vengono ritornate 765,164 righe e per flight_2_3 2,401,092.

3. **Merge Interval**: si tratta di un operatore interessante in quanto ha il compito di riconoscere a tempo di esecuzione se due o più intervalli sono tra loro sovrapposti e nel caso in cui lo siano li traduce in intervalli che non si sovrappongono; nell'ambito dello

studio delle performance rappresenta un operatore importante che può influenzare positivamente le prestazioni, perché permette di evitare la lettura ripetuta dei dati. Nel caso preso in esame Merge Interval è sfruttato contestualmente alla realizzazione dell'index Seek Dinamico, per determinare dei range non sovrapposti a partire dal predicato di join, sulla base dei quali realizzare poi la ricerca; lo stesso operatore però viene impiegato da SQL Server anche in altri ambiti, per esempio relativamente a query che impongono nella clausola WHERE dei predicati costruiti in base ai costrutti BETWEEN o IN, in cui i valori sono rappresentati mediante variabili. In questo caso infatti SQL Server non è in grado di stabilire a tempo di compilazione se gli intervalli definiti nel predicato sono sovrapposti o meno, pertanto ricorre all'operatore Merge Interval. Anche in questo caso quindi SQL Server introduce un operatore decisamente utile al fine di migliorare le prestazioni.

Oracle XE prevede la stessa logica di base di MS SQL Server per la versione senza COALESCE, però vi sono delle differenze sostanziali:

1. **Viste intermedie**: Oracle XE sfrutta diverse viste intermedie, laddove SQL Server utilizza Table Spool; anche in questo caso si tratta di un'operazione che influisce positivamente sulle prestazioni, perché è più conveniente recuperare i dati da una vista, piuttosto che dover eseguire nuovamente le operazioni che li hanno determinati. Si può notare che Oracle fa uso di diverse viste intermedie.
2. **Non determina dei range per Index Seek:** vengono determinate dinamicamente a tempo di esecuzione le righe che fanno match all'interno di un join anche quando la condizione di join imposta è complessa perché espressa mediante OR; in questo caso quindi, a differenza di SQL Server, non sono presenti range determinati dinamicamente da controllare, ma è comunque possibile realizzare l'operazione a tempo di esecuzione analizzando la tabella a livello di riga. In realtà la tabella non è scorsa staticamente mediante Full Table Access, ma grazie alle operazioni di Index Unique Scan e Table Access by ROWID, che permettono di accedere solo alle righe realmente coinvolte nel join, la cui join key fa match, risparmiando quindi tempo e accessi in I/O al disco.
3. **Non prevede Merge Interval**, in quanto, come detto, non determina range dinamici.

Quindi sia Oracle XE, sia MS SQL Server realizzano Index Seek dinamici, con la differenza che SQL Server basa l'operazione sulla definizione a tempo di esecuzione di range non sovrapposti da controllare, mentre Oracle XE non opera a livello di range, ma a livello di singola riga.

Tornando a SQL Server, le versioni che prevedono COALESCE, nonostante gli operatori ora citati, registrano comunque un tempo di esecuzione nettamente inferiore; questo è determinato anche dal fatto che spesso si ricorre alla tecnica del parallelismo, seconda macro differenza che intercorre tra SQL Server, PostgreSQL ed Oracle XE.
Il Query Optimizer di SQL Server infatti ha la possibilità di realizzare due tipi di piani di esecuzione: seriali nel caso in cui vi sia un unico thread per l'esecuzione della query, e paralleli, nel caso in cui invece si faccia ricorso a più threads per eseguire le operazioni necessarie. E' importante sottolineare che il numero di threads coinvolti è riferito alla singola operazione e non

al piano di esecuzione complessivo.
Quando viene superata la soglia impostata come Cost Threshold for Parallelism[13] il Query Optimizer genera dei piani di esecuzione paralleli, che prevedono l'introduzione degli Exchange operators, ovvero nuove operazioni per gestire il parallelismo (Distribute Streams, Repartition Streams, Gather Streams) che hanno ovviamente un costo associato; qualora il costo complessivo relativo al piano parallelo risulti inferiore a quello associato al piano seriale, sarà privilegiato il primo e questo è ciò che solitamente accade; in altre circostanze, tuttavia, SQL Server, nonostante sia stato superato il valore di Cost Threshold for Parallelism, può continuare ad optare per il piano seriale; questo in genere accade, come per il piano mostrato in Figura 6, se i benefici derivanti dal parallelismo sono inferiori rispetto all'aumento di costo apportato dalle nuove operazioni introdotte per la gestione del parallelismo, oppure se si verificano errori in merito alla stima della cardinalità delle relazioni, dovute al fatto che non sono fornite al query optimizer delle informazioni accurate sulle quali basare le proprie stime.
Soffermandosi poi sugli Exchange operators, si può innanzitutto notare dalla Figura 11 che l'operatore di Repartition Streams risulta particolarmente intelligente, essendo in grado di mantenere l'ordinamento preesistente delle righe; ovviamente questo aspetto riveste grande importanza perché permette successivamente di realizzare comunque un Merge Join.
In generale gli Exchange Operators permettono di realizzare il parallelismo e la differenza sostanziale tra un piano seriale e l'analogo parallelo è che nel primo vi è un singolo thread che legge i dati da una sorgente dati, li processa tramite un certo numero di operatori necessari e infine ritorna i risultati a destinazione; in un piano parallelo un singolo thread legge ugualmente i dati da una sorgente dati, li processa tramite gli operatori necessari a tal fine e ritorna i risultati a destinazione, ma in questo vi è una sostanziale differenza: la destinazione è un Exchange operator, così come lo può essere la sorgente.
In generale comunque sfruttare la tecnica del parallelismo significa diminuire i tempi di esecuzione e quindi aumentare le prestazioni; al tempo stesso si registra un aumento del tempo di CPU.
Questo risulta particolarmente evidente nei due casi esaminati:

- In relazione alla query di Figura 11, il costo associato al piano **parallelo è 216,806** mentre quello associato all'analogo **piano seriale è 230,375;** ciò comporta un aumento del tempo di esecuzione per la versione seriale di 1 secondo.
- In relazione alla query di Figura 15, il costo associato al piano parallelo è 212.71 mentre quello del piano seriale è 227.181; l'aumento del tempo di esecuzione del piano seriale è di 4 secondi.

Analizzando i costi degli operatori dei piani seriali e i relativi piani paralleli si può inoltre notare che proprio l'adozione del parallelismo consente una diminuzione dei costi di ogni operatore, ed in particolare le operazioni che maggiormente ne beneficiano sono le operazioni di join, il cui

[13] Cost-Threshold for Parallelism: opzione di configurazione di SQL Server che consente di specificare la soglia in corrispondenza della quale MS SQL Server crea ed esegue piani paralleli per le query. In SQL Server viene creato ed eseguito un piano parallelo solo quando il costo stimato per l'esecuzione di un piano seriale per la stessa query è superiore al valore impostato nell'opzione. Il costo equivale al tempo (in secondi) stimato per l'esecuzione del piano seriale in una configurazione hardware specifica. Di default è impostato a 5. [16]

costo viene drasticamente diminuito, fattore che ovviamente concorre alla riduzione del costo globale associato alla query.
Volendo poi individuare quali possono essere state le cause che hanno portato all'adozione del parallelismo in seguito ad un aumento del costo complessivo associato alla query, nei due casi riportati si possono dedurre due motivazioni: per quanto riguarda il primo esempio si può affermare che sia risultato più conveniente adottare un piano parallelo in quanto entrambe le relazioni coinvolte nell'Hash Match sono di dimensioni particolarmente elevate (2,648,963 e 6,351,272 righe); fattore che ha portato ad un aumento del costo dell'operatore, che si è ritrovato a dover processare un numero maggiore di righe e soprattutto a creare la hash table in memoria a partire da una tabella di dimensioni maggiori - operazione, questa, onerosa; dall'ultimo esempio si può invece intuire che il fattore determinante che ha spinto all'utilizzo del piano parallelo è il fatto che sia stata imposta una condizione sulla relazione più grande, che comprende 6,351,272 righe, ed il relativo filtro è stato applicato direttamente in concomitanza con l'operatore Clustered Index Scan.
Imporre un filtro direttamente sulla tabella non è un'operazione particolarmente gravosa, ma il lievissimo aumento di CPU time speso per analizzare ogni riga, oltre che semplicemente scorrerle e ritornarle, deve essere moltiplicato per tutte le 6,351,272 righe della relazione su cui è applicato il predicato. Si fa in questo modo consistente l'extra CPU time e ovviamente diventa influente sul costo dell'operatore; per diminuire i tempi di esecuzione viene quindi scelto il piano parallelo.
Tale ipotesi ha poi trovato conferma dal fatto che applicando la condizione alla relazione di dimensioni minori si è tornati ad un query plan seriale.
Per quanto riguarda invece PostgreSQL, esso è *process-based* e non *threaded-based*, il che significa che utilizza un unico processo per ogni database session. Ne deriva che ogni singola connessione al database non può utilizzare più di una CPU. Ovviamente però se sono presenti sessioni multiple queste saranno suddivise tra le CPU disponibili dal sistema operativo.
Anche per Oracle XE non è previsto il parallelismo. Infatti anche se Oracle XE è installato su un computer che ha più di una CPU a disposizione, consumerà sempre al massimo le risorse di una singola CPU. Per poter sfruttare interamente le risorse di elaborazione del computer è necessario eseguire l'upgrade alle versioni Oracle Database Standard o Enterprise Edition.
Proprio per questo motivo, anche se di fronte ad una query molto complessa e che richiede un utilizzo intensivo della CPU, PostgreSQL e Oracle non sono in grado di sfruttare più di una CPU per l'esecuzione della query.

Un altro elemento importante ai fini dell'analisi delle prestazioni, legato ai piani paralleli, che ha implementato SQL Server e che non si ritrova ovviamente né in PostgreSQL né in Oracle XE, è l'operatore Bitmap Filter.
Innanzitutto si tratta di un operatore presente solo nei query plan paralleli, il cui scopo è quello di velocizzarli rimuovendo le righe di un sottoalbero non necessarie quanto prima nella query; gli operatori successivi hanno così meno righe su cui lavorare e le prestazioni complessive migliorano.
Nel caso esaminato (Figura 15) il filtro viene imposto prima che le righe siano passate all'operatore

di Hash Join. In particolare il filtro è creato sul lato del build input[14], come si può notare anche dal piano di esecuzione indicato, ma in realtà il controllo che si occupa di filtrare le righe è applicato contestualmente all'operatore di parallelismo che si trova sul lato del probe input[15].
Internamente i filtri bitmap sono implementati come semplici array di bit. Quando viene costruito un filtro bitmap è assegnato un hash value ad ogni riga del lato del build input e vengono poi settati i bit nell'array che corrispondono agli hash value che contengono almeno una riga. Quando poi si controlla per il filtraggio la tabella nel lato di probe, ancora una volta si genera un valore di hash per ogni riga e si controlla se il bit relativo nell'array è settato o meno; se non è settato si sa immediatamente che la riga non rientrerà nel join, non essendoci match quindi con l'altra tabella di build input, e la riga viene scartata; in questo modo si realizza una rapida eliminazione delle righe non interessate nel join, diminuendo la complessità e il costo del successivo inner join.
Un aspetto da notare è che l'operatore di Bitmap Filter non viene utilizzato sempre all'interno di un piano parallelo in cui compare un'operazione di Hash Join; ovviamente si tratta di una decisione presa dal Query Optimizer se decreta che il filtro è abbastanza selettivo da risultare effettivamente utile. Inoltre solitamente si fa ricorso a questo operatore in presenza di un Hash Join, ma può capitare anche che venga impiegato in un Merge Join.

Un ulteriore aspetto che differenzia fortemente SQL Server da una parte e PostgreSQL e Oracle XE dall'altra, è che il primo crea un Clustered Index sulla primary key, mentre i secondi generano un indice sulla chiave primaria non di tipo clustered, ordinato in quanto realizzato mediante b-tree[16].
Gli indici cluster ordinano e archiviano le righe di dati della tabella in base ai valori di chiave, ovvero alle colonne incluse nella definizione dell'indice. Per ogni tabella è disponibile un solo indice cluster, poiché alle righe di dati è possibile applicare un solo tipo di ordinamento. Dalla definizione quindi si ricava che con un indice di tipo Cluster le righe sono fisicamente memorizzate su disco nello stesso ordine dell'indice e questo è il motivo per cui per ogni tabella è possibile avere un solo indice Cluster; sono più veloci nel caso di lettura proprio perché i dati sono memorizzati in maniera ordinata [6].
Gli indici non cluster presentano una struttura distinta dalle righe di dati. Un indice non cluster contiene i valori di chiave dell'indice non cluster, ciascuno dei quali dispone di un puntatore alla riga di dati che contiene il valore di chiave [6]. In questo caso quindi vi è una seconda lista ordinata che contiene un duplicato della colonna indicizzata che ha puntatori alle righe fisiche; questo fa sì che se si vuole accedere ai dati che non compaiono nell'indice tramite indice non cluster bisogna prima scorrere l'indice e per ogni valore ritornare i dati richiesti, raggiungibili mediante il

[14] Il build input di un Hash Join è solitamente costituito dalla tabella di dimensioni minori, in quanto su di essa viene applicata una *hashing function* per la creazione di una *hash table*; tale funzione lavora sulla join key e determina l'indice al quale la riga analizzata deve essere memorizzata.
[15] È solitamente scelta come probe input di un Hash Join la relazione di dimensioni maggiori. Sulla sua join key viene riapplicata la *hashing function* per determinare la posizione delle righe nella *hash table*; se nelle posizioni determinate sono presenti delle righe e queste fanno match con le righe di partenza, il join è eseguito.
[16] Struttura dati che permette la rapida localizzazione dei file. Ogni chiave appartenente al sottoalbero sinistro di un nodo è di valore inferiore rispetto ad ogni chiave appartenente ai sottoalberi alla sua destra. Inoltre è bilanciato: per ogni nodo le altezze dei sottoalberi destro e sinistro differiscono di al più un'unità; questo permette di eseguire operazioni di inserimento, cancellazione e ricerca in tempi ammortizzati logaritmicamente.

puntatore associato. Viene così introdotto un livello indiretto per l'accesso ai dati e perciò risulta più lenta l'operazione di lettura.
Da questo aspetto si può comprendere come un indice di tipo cluster, diversamente dai non clustered, spesso permette di ridurre le operazioni di I/O da disco, specialmente se la tabella interessata è di grandi dimensioni.
Si possono però avere per una stessa tabella più indici non cluster. Risultano più veloci nelle operazioni di inserimento e aggiornamento [7].
Le query esaminate precedentemente effettuavano solo delle letture e non inserimenti o aggiornamenti. In questo ambito pertanto risulta vincente la scelta di SQL Server.
Inoltre un importante aspetto da sottolineare è che SQL Server, avendo i dati fisicamente ordinati sulla base della primary key, spesso impiega come tecnica di join il Merge Join. Infatti, nelle query precedenti, in cui venivano coinvolte tre relazioni nei Full Outer Join, i piani di esecuzione relativi presentavano due join distinti, di cui il primo era sempre un Merge Join; in questo modo ovviamente viene sfruttato l'ordinamento già esistente delle relazioni e può essere utilizzato il Merge Join che è la tecnica più efficiente nel caso in cui vi sia un ordinamento sulla chiave di join. In un caso anche il secondo join è stato realizzato mediante Merge Join (Figura 19), perché è eseguito tra una relazione già ordinata in base alla clustered primary key e una relazione di piccole dimensioni (180,357 righe), il cui ordinamento quindi è risultato poco dispendioso.
PostgeSQL e Oracle XE al contrario, non avendo a disposizione indici cluster sulla chiave primaria, in tutti gli esempi riportati, prima realizzano entrambi i join mediante Hash Match, che rappresenta in un certo senso l'ultima carta da giocare quando lo scenario non presenta delle caratteristiche che possano favorire le altre due tipologie di join; in questo modo le prestazioni di PostgeSQL e Oracle XE risultano inferiori.
Ovviamente non viene sfruttato in questo caso il fatto che l'indice non cluster sia ordinato, perché nella SELECT delle query viene richiesto di ritornare tutti i campi e non solo la chiave primaria. In questo modo, quindi, viene sempre realizzato un Hash Match, piuttosto di un Merge Join che si sarebbe basato sull'ordinamento logico dell'indice, in quanto quest'ultimo avrebbe comportato un incremento del costo. Tale incremento sarebbe stato determinato dalla presenza di un ulteriore passaggio intermedio per poter appunto sfruttare l'ordinamento logico: l'operazione di Merge Join avrebbe infatti dovuto attraversare il b-tree dell'indice e per ogni singola entry accedere mediante l'apposito puntatore ai dati effettivi memorizzati in memoria.

Un elemento che può influire positivamente sulle performance comune sia a SQL Server sia a PostreSQL è l'utilizzo dell'operatore Filter. In particolare dalle query eseguite in Figura 19 e Figura 20 si può notare come entrambi, nel caso in cui vi siano più condizioni espresse nella clausola WHERE, di cui una molto selettiva e le altre non particolarmente, optino per applicare subito il filtro più restrittivo nell'ambito dell'operatore che scorre le righe, mentre lasciano come ultima operazione prima della SELECT l'applicazione dei filtri rimanenti.
Per le prestazioni si tratta sicuramente di un'operazione vantaggiosa, perché permette di eliminare in partenza, tramite il predicato altamente selettivo, un numero elevato di righe, ben 1,826,090. Tale eliminazione comporta importanti benefici, tra i quali il fatto che le operazioni successive possono lavorare con un numero fortemente ridotto di righe - e ciò comporta una

diminuzione del costo loro associato e del tempo di esecuzione da loro impiegato - e la possibilità per SQL Server di eseguire due Merge Join, meno costosi.
La tecnica ora esposta rappresenta lo scenario migliore per ottenere prestazioni superiori, infatti:

- Se SQL Server e PostgreSQL avessero applicato subito tutte le condizioni: sarebbe aumentato il costo associato agli operatori di Clustered Index Scan per SQL Server e di Sequence Scan per PostgreSQL, dovuto al fatto che avrebbero dovuto scorrere e in più analizzare ogni singola riga -ricordando che si tratta delle tabelle di dimensioni maggiori-, si sarebbero quindi dovute analizzare molte più righe perché non si sarebbe sfruttata la selezione operata dai Merge Join nel caso di SQL Server e degli Hash Match nel caso di PostgreSQL. Questo tra l'altro non avrebbe portato ad un effettivo vantaggio perché soltanto una condizione è fortemente selettiva, quindi si avrebbe avuto l'aumento del costo detto, non bilanciato poi da un effettivo vantaggio, in quanto il numero di righe rimosse dai filtri poco selettivi sarebbe stato basso.
- Se avessero applicato tutte le condizioni come penultima operazione: il costo di Clustered Index Scan sulla tabella su cui è applicato il filtro più restrittivo per SQL Server e di Sequence Scan per PostgreSQL sarebbe stato inferiore, ma sarebbe aumentato quello delle operazioni successive, che si sarebbero ritrovate a dover elaborare un numero decisamente superiore di righe, sia perché non scremate immediatamente dal filtro, sia perché i due join sarebbero stati dei Full Outer Join e non dei Left/Right Join, quindi non avrebbero eliminato ulteriori righe della relazione di destra/sinistra; inoltre per SQL Server probabilmente non sarebbe stato eseguito un Merge Join per unire i risultati dei due sottoalberi, ma un ben più oneroso Hash Match.

In questo contesto l'unica differenza tra SQL Server e PostgreSQL è che il primo prevede un operatore a parte dedicato che si può vedere nel piano di esecuzione (operatore Filter), mentre PostgreSQL lo ingloba direttamente nell'ultimo Hash Match prima della SELECT; la sostanza rimane comunque invariata in quanto in entrambi i casi i filtri vengono applicati su 180,357 righe e ne eliminano 69,604.
Per quanto riguarda Oracle XE, anche in questo caso è prevista la possibilità di non applicare direttamente tutte le condizioni nel caso non siano restrittive, e come per PostgreSQL questo non avviene mediante un'operazione a sé stante, ma in concomitanza con l'operatore View, come si vede in Figura 21.
Vi è però una differenza rispetto il comportamento adottato da SQL Server e PostgreSQL, ovvero Oracle XE ricade nella situazione descritta precedentemente, nella quale tutte le condizioni sono applicate prima della SELECT, come penultima operazione. Valgono pertanto le considerazioni espresse in merito, per cui si registra un minor costo associato alle operazioni Full Access Table ma, non potendo godere dell'eliminazione delle righe della condizione più restrittiva, si ha un maggior costo per le successive operazioni che impiegano quindi più tempo per terminare; inoltre i join realizzano le operazioni logiche di Full Outer Join. Per confermare quanto detto si riportano alcuni dati, ricavati da Figura 21:

1. *Table Access (Full) su flight_2_3 senza applicare la condizione*:
    a. Costo: 13575
    b. CPU time: 00:02:43

2. *Hash Join (Full Outer) tra flight_2_3 non filtrata e flight_1_1:*
    a. Costo: 44501
    b. CPU Time: 00:08:55
3. *Table Access (Full) su flight_2_3 applicando la condizione*:
    a. Costo: 13584
    b. CPU time: 00:02:44
4. *Hash Join (Full Outer) tra flight_2_3 filtrata e flight_1_1*:
    a. Costo: 44372
    b. CPU Time: 00:08:53

Quello appena esposto si configura pertanto come uno degli elementi determinanti nel fatto che, per quanto riguarda questa specifica query, Oracle XE abbia dei tempi di esecuzione più elevati - più del doppio - rispetto PostgreSQL, nonostante presentino dei piani di esecuzione del tutto analoghi.

7.1.1.6 Comparazione con MySQL

Per quanto riguarda MySQL si può innanzitutto sottolineare ulteriormente il fatto che si tratta dell'unico RDBMS considerato che non supporta l'operazione di Full Outer Join.
Altra grande limitazione rispetto gli altri RDBMS esaminati, che ne riduce le prestazioni, è la mancanza del supporto di algoritmi di join diversi dal Nested Loop Join; MySQL, infatti, è in grado di realizzare join soltanto applicando la tecnica del Nested Loop Join. Questo ovviamente porta a scarse prestazioni nei casi in cui questo non sia l'algoritmo maggiormente indicato.
Il Nested Loop Join indicizzato, cioè realizzato su indici, ha infatti delle prestazioni migliori rispetto Merge Join e Hash Join solo quando le dimensioni delle tabelle coinvolte sono ridotte, altrimenti è raramente la scelta ottimale.
MySQL comunque adotta delle piccole accortezze per rendere maggiormente efficiente il Nested Loop a tempo di esecuzione, infatti in tutti i casi considerati i Nested Loop sono realizzati in modo da individuare le sole righe dell'inner input per le quali esiste effettivamente il match con l'outer input (condizioni di join verificate), evitando di scorrere tutte le entry dell'inner input, ma prelevando le sole righe interessate, risparmiando in questo modo accessi al disco. Quindi per ogni riga dell'outer input, viene prelevata la chiave di join e utilizzata, tramite l'operazione di Unique Key Lookup, come punto di accesso sull'indice dell'inner input costruito sulla chiave di join, viene quindi prelevata l'intera riga corrispondente; se vi è un match ovviamente la riga è accettata nel join. Questo è possibile solo nel caso in cui il join venga realizzato sulle chiavi primarie, su cui MySQL costruisce automaticamente un indice, o altri campi su cui è stato definito un indice (Nested Loop indicizzati).
Inoltre, sempre per incrementare l'efficienza dell'algoritmo di Nested Loop, MySQL applica l'operazione di Full Table Scan, necessaria per la determinazione dell'outer input, sulla tabella di dimensioni minori, in modo da ridurne il costo.
Approfondendo l'analisi, inoltre, si può notare che anche MySQL non supporta la tecnica del parallelismo, e quindi ogni query risulta sempre *single-threaded*; ciò implica che anche nei casi in

cui la query risulti particolarmente complessa, essa viene sempre eseguita sfruttando un unico processo, e ciò spesso determina dei tempi di esecuzione anche piuttosto elevati, come si può infatti notare dai dati riportati. Questo aspetto accomuna MySQL a PostgreSQL e Oracle EX.
Un aspetto che invece lo accomuna a SQL Server è che anche MySQL prevede la creazione automatica di indici di tipo Cluster sulla primary key; infatti quando viene definita una primary key su una tabella, InnoDB[17] la utilizza come indice cluster, e da ciò derivano tutti i vantaggi elencati in precedenza in termini di prestazioni. In realtà, differentemente da SQL Server che sfrutta l'ordinamento sulla clustered primary key per realizzare delle operazioni di Merge Join, MySQL non è in grado di eseguire Merge Join, per cui vede notevolmente ridotti i vantaggi di cui potrebbe beneficiare grazie all'indice cluster.
Per quanto riguarda i filtri, MySQL valuta le clausole WHERE direttamente a livello di tabella, tramite l'operazione di Full Table Scan, se il filtro è applicato sull'inner input di un Nested Loop Join, o tramite l'operazione di Unique Key Lookup se applicato sull'outer input. In questo modo il costo delle operazioni citate aumenta. MySQL quindi, a differenza di SQL Server, PostgreSQL e Oracle, non dispone di un'apposita operazione di filtraggio, che come già affermato, può portare dei vantaggi in termini di prestazioni, perché permette di evitare l'aumento del costo associato all'operatore che deve applicare il predicato oltre a svolgere la sua funzione "standard" (in questo caso il Nested Loop).
Sono state infatti riproposte le stesse condizioni che erano state applicate anche nelle query di partenza lanciate sugli altri DBMS, e mentre gli altri tre DBMS sfruttavano l'operazione di filtraggio in un momento successivo alle operazioni di scan, MySQL applica sempre i filtri immediatamente, senza sfruttare la riduzione delle righe apportata dai join, indipendentemente dalla natura selettiva o meno dei predicati.
Mancando il supporto per il Full Outer Join non è possibile eseguire un confronto adeguato con gli altri RDBMS presi in analisi, che invece la supportano.
Per realizzare quindi un confronto più equo, sono state immesse le stesse query elaborate per MySQL anche negli altri RDBMS. Vengono sotto riportate le query più interessanti di quelle proposte precedentemente, con i relativi tempi e piani di esecuzione prodotti da SQL Server, PostgreSQL e Oracle.
Per tutte vengono inoltre riportati i tempi di esecuzione, per un confronto più completo.

[17] InnoDB è uno Storage Engine di MySQL, fornito in tutte le sue distribuzioni.

QUERY 1, NESSUNA CONDIZIONE APPLICATA.

MS SQL SERVER

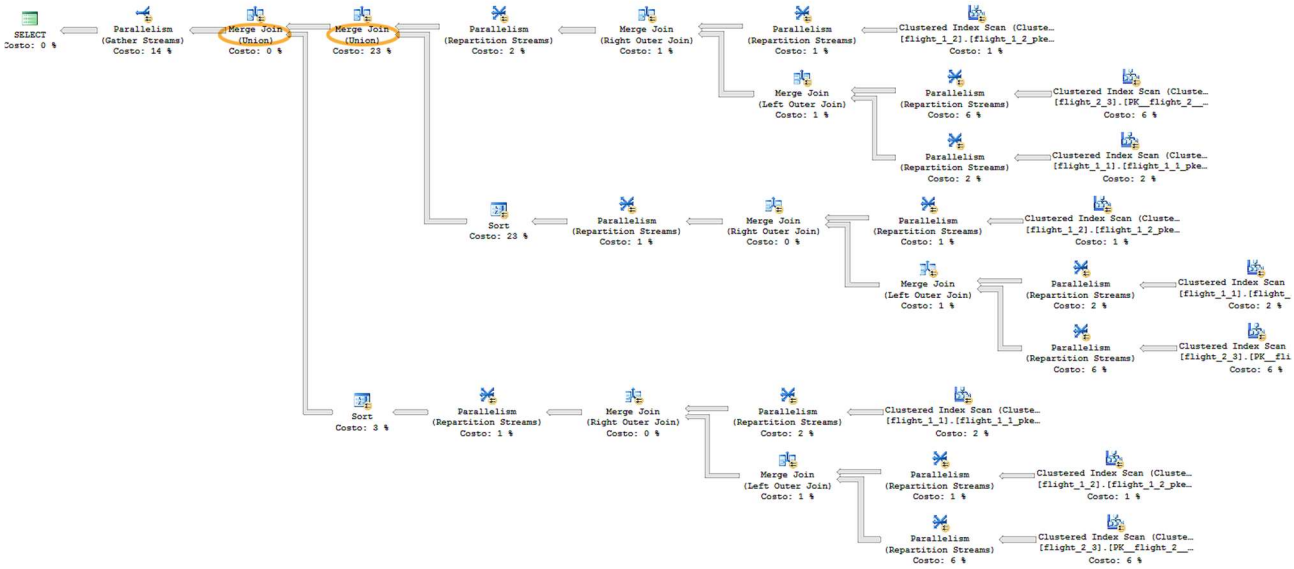


Figura 25

Come per MySQL viene mantenuta la struttura che prevede la ripartizione in 3 blocchi, ma SQL Server presenta delle differenze notevoli rispetto a MySQL, che infatti portano ad una drastica riduzione dei tempi di esecuzione:

- **Ogni operazione è eseguita in parallelo**, quindi SQL Server sfrutta 8 threads che lavorano contemporaneamente, spartendosi le righe da processare; questo significa che nel caso peggiore, ovvero quello in cui l'operatore è applicato su ben 6,351,271 righe, ogni thread elabora in media 700,000 delle righe totali, come si ricava analizzando i dati del query plan.
  Ovviamente questo si traduce in un notevole incremento delle prestazioni in quanto abbatte i tempi di esecuzione.
- **I join vengono realizzati mediante Merge Join**, in quanto SQL supporta anche questa tipologia, che gli permette in questo caso di sfruttare il fatto che le relazioni sono già ordinate sulla Clustered Primary Key e quindi è in grado di realizzare un join meno oneroso rispetto il Nested Loop previsto da MySQL.
  Infatti comparando i due piani di esecuzione si nota come entrambi i DBMS presentino la stessa struttura: in ognuno dei tre blocchi in cui sono suddivisi i query plan, viene prima eseguito un Left Outer Join tra le prime due relazioni che compaiono nel campo FROM, e in seguito il risultato di questo è unito mediante un Right Outer Join con la terza tabella rimanente.
  La differenza è insita nel fatto che MySQL realizza i Left/Right Outer join mediante Nested Loop Join e SQL Server, invece, mediante Merge Join. La tecnica di SQL Server risulta quella più indicata perché sfrutta l'ordinamento già presente e

mantenuto dagli operatori di Repartition Streams, risultando quindi più conveniente.

- **L'operazione di UNION è realizzata mediante due Merge Join**, eseguiti consecutivamente, per unire i risultati derivati dai tre blocchi. Queste operazioni richiedono di riordinare le righe provenienti dai rispettivi blocchi, mediante l'operatore Sort che opera sui campi id.
  È interessante soffermarsi su come SQL Server realizzi l'operazione di UNION: l'operazione logica ha un unico output e 3 child input e dopo l'ottimizzazione operata dal query optimizer l'albero del query plan relativo all'operazione di UNION sarebbe costituito da Merge Union con tre input; il query optimizer supporta operazioni n-arie, mentre il motore di esecuzione non le supporta, per cui l'output prodotto dal query optimizer viene rielaborato in una forma gestibile dal motore di esecuzione, suddividendo l'operazione di Merge Union a tre vie in due operazioni di Merge Join consecutive; questo in particolare è visibile sul piano di esecuzione in cui il costo associato all'operazione "originale" è inserito solo su una delle due operazioni di Merge Union, mentre l'altra ha il costo associato settato a zero.

POSTGRESQL

```
1  QUERY PLAN
2  Unique  (cost=4947834.78..5407175.23 rows=9186809 width=161)
3    ->  Sort  (cost=4947834.78..4970801.81 rows=9186809 width=161)
4        Sort Key: f23.id, f23.deptime, f23.crsdeptime, f23.arrtime, f23.crsarrtime, f11.id, f11.uniquecarrier, f11.flightnum,
   f11.tailnum, f11.origin, f11.dest, f11.distance, f12.id, f12.uniquecarrier, f12.flightnum,
   f12.tailnum, f12.origin, f12.dest, f12.distance
5        ->  Append  (cost=1.53..1624510.31 rows=9186809 width=161)
6              ->  Merge Left Join  (cost=1.53..553500.71 rows=6351328 width=161)
7                    Merge Cond: (f23.id = f12.id)
8                    ->  Merge Left Join  (cost=1.11..479001.32 rows=6351328 width=104)
9                          Merge Cond: (f23.id = f11.id)
10                         ->  Index Scan using flight_2_3_pkey on flight_2_3 f23  (cost=0.56..330887.44 rows=6351328 width=48)
11                         ->  Index Scan using flight_1_1_pkey on flight_1_1 f11  (cost=0.55..107154.97 rows=2006447 width=56)
12                   ->  Index Scan using flight_1_2_pkey on flight_1_2 f12  (cost=0.42..48258.15 rows=829034 width=57)
13             ->  Hash Left Join  (cost=124314.32..523564.23 rows=2006447 width=161)
14                   Hash Cond: (f11_1.id = f12_1.id)
15                   ->  Hash Right Join  (cost=87375.06..399202.45 rows=2006447 width=104)
16                         Hash Cond: (f23_1.id = f11_1.id)
17                         ->  Seq Scan on flight_2_3 f23_1  (cost=0.00..128765.28 rows=6351328 width=48)
18                         ->  Hash  (cost=42699.47..42699.47 rows=2006447 width=56)
19                               ->  Seq Scan on flight_1_1 f11_1  (cost=0.00..42699.47 rows=2006447 width=56)
20                   ->  Hash  (cost=17670.34..17670.34 rows=829034 width=57)
21                         ->  Seq Scan on flight_1_2 f12_1  (cost=0.00..17670.34 rows=829034 width=57)
22             ->  Hash Right Join  (cost=168485.18..455577.28 rows=829034 width=161)
23                   Hash Cond: (f23_2.id = f12_2.id)
24                   ->  Seq Scan on flight_2_3 f23_2  (cost=0.00..128765.28 rows=6351328 width=48)
25                   ->  Hash  (cost=143549.25..143549.25 rows=829034 width=113)
26                         ->  Hash Right Join  (cost=36939.26..143549.25 rows=829034 width=113)
27                               Hash Cond: (f11_2.id = f12_2.id)
28                               ->  Seq Scan on flight_1_1 f11_2  (cost=0.00..42699.47 rows=2006447 width=56)
29                               ->  Hash  (cost=17670.34..17670.34 rows=829034 width=57)
30                                     ->  Seq Scan on flight_1_2 f12_2  (cost=0.00..17670.34 rows=829034 width=57)
```

Figura 26

Ancora una volta è presente la suddivisione del query plan nei tre blocchi principali, uniti poi in ultimo mediante l'operazione di UNION.

Le differenze che si possono riscontrare rispetto la versione di MySQL, sono:

- Realizzazione dei join mediante sia Hash Join, sia Merge Join. Anche PostgreSQL supporta diversi algoritmi per i join, differentemente da MySQL, e questo denota subito una maggiore completezza a vantaggio di PostgreSQL.
  Dal piano di esecuzione si può notare come PostgreSQL preferisca realizzare un Merge Join, più costoso rispetto un Hash Join, qualora sia necessario per agevolare una successiva operazione di Sort.
- L'operazione di UNION è realizzata in due passaggi consecutivi: i risultati dei 3 blocchi vengono dapprima concatenati (APPEND), in seguito ordinati e infine viene applicato l'operatore UNIQUE che si occupa di eliminare eventuali duplicati.

ORACLE

Operation	Object	Optimizer	Cost	Cardinality	Bytes
SELECT STATEMENT		ALL_ROWS	548,710	9,228,460	1,578,066,660
SORT (UNIQUE)			548,710	9,228,460	1,578,066,660
UNION-ALL			0	0	0
HASH JOIN (RIGHT OUTER)			86,251	6,351,271	1,086,067,341
TABLE ACCESS (FULL)	FLIGHT_1_1	ANALYZED	4,839	2,006,446	120,386,760
HASH JOIN (RIGHT OUTER)			37,613	6,351,271	704,991,081
TABLE ACCESS (FULL)	FLIGHT_1_2	ANALYZED	2,002	829,034	48,913,006
TABLE ACCESS (FULL)	FLIGHT_2_3	ANALYZED	13,575	6,351,271	330,266,092
HASH JOIN (RIGHT OUTER)			61,234	2,035,960	348,149,160
TABLE ACCESS (FULL)	FLIGHT_1_2	ANALYZED	2,002	829,034	48,913,006
HASH JOIN (OUTER)			44,501	2,035,960	228,027,520
TABLE ACCESS (FULL)	FLIGHT_1_1	ANALYZED	4,839	2,006,446	120,386,760
TABLE ACCESS (FULL)	FLIGHT_2_3	ANALYZED	13,575	6,351,271	330,266,092
HASH JOIN (OUTER)			54,187	841,229	143,850,159
HASH JOIN (OUTER)			37,613	841,229	93,376,419
TABLE ACCESS (FULL)	FLIGHT_1_2	ANALYZED	2,002	829,034	48,913,006
TABLE ACCESS (FULL)	FLIGHT_2_3	ANALYZED	13,575	6,351,271	330,266,092
TABLE ACCESS (FULL)	FLIGHT_1_1	ANALYZED	4,839	2,006,446	120,386,760

Figura 27

Persiste la suddivisione nei tre blocchi principali poi uniti mediante UNION.
Le differenze riscontrate rispetto MySQL sono:

- Join realizzati mediante Hash Join: Oracle XE è in grado di sfruttare i tre principali algoritmi di join, quindi applica quello più conveniente in base alla situazione. In questo caso opta per un Hash Join in quanto i dati non sono ordinati sulla chiave primaria (indice sulla primary key non clustered) e tutte le tabelle da unire mediante join hanno dimensioni elevate, pertanto sarebbe sconveniente il Nested Loop.
- L'operazione di UNION è realizzata mediante UNION ALL e Sort (Unique): UNION ALL è un operatore che accetta i set di righe prodotti dai blocchi e ritorna l'unione dei set, senza eliminare i duplicati. Sort Unique si occupa invece di ordinare i risultati dell'unione e contemporaneamente elimina eventuali duplicati.

Il piano di esecuzione di Oracle è simile quindi a quello generato da PostgreSQL; ma risulta più lento di 58 secondi.
La differenza principale tra i piani di esecuzione prodotti da SQL Server, PostgreSQL e Oracle XE risiede nelle differenti modalità secondo le quali viene realizzata l'operazione di UNION, alla quale si aggiunge ovviamente il fatto che PostgreSQL e Oracle XE non supportano il parallelismo.

QUERY 3, 2 CONDIZIONI APPLICATE

```
SELECT f23. * , f11. * , f12. *
FROM flight_2_3 f23
LEFT OUTER JOIN flight_1_1 f11 on f23.id=f11.id
LEFT OUTER JOIN flight_1_2 f12 on f23.id=f12.id
WHERE f23.deptime>'1500'
AND f12.flightnum>'1000'
UNION
SELECT f23. * , f11. * , f12. *
FROM flight_1_1 f11
LEFT OUTER JOIN flight_2_3 f23 on f23.id=f11.id
LEFT OUTER JOIN flight_1_2 f12 on f12.id=f11.id
WHERE f23.deptime>'1500'
AND f12.flightnum>'1000'
```

SQL SERVER

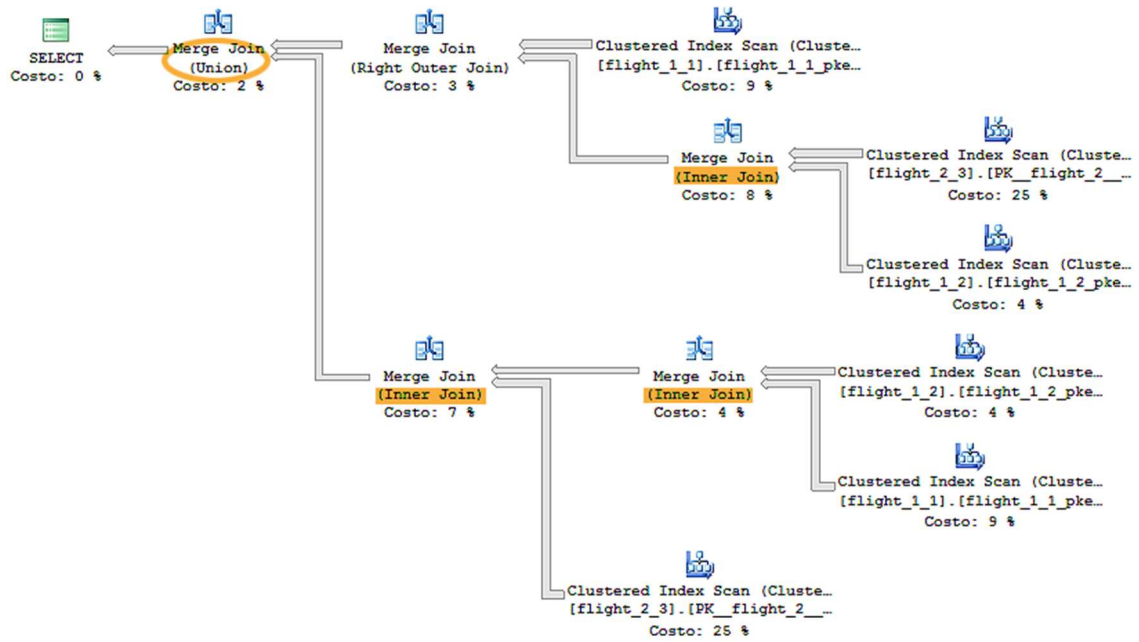


Figura 28

Rimangono valide le considerazioni precedenti in relazione al confronto con MySQL; anche in questo caso viene ricalcata la struttura adottata da MySQL stesso con la suddivisione del query plan in due blocchi principali. Le differenze che si possono riscontrare tra MySQL e SQL Server sono le stesse già elencate in merito al confronto precedente con l'eccezione che stavolta è stato scelto un piano seriale, sia per la presenza di Inner Join in sostituzione degli Outer Join precedenti, che sono meno dispendiosi, sia perché il numero di righe coinvolto è diminuito.
Ovviamente, essendo ora presenti solo due blocchi, la UNION è realizzata mediante un solo

operatore Merge Union, non essendo infatti più necessario tradurre una UNION a tre vie come accadeva invece nel caso precedente.

POSTGRESQL

```
1  QUERY PLAN
2  Unique  (cost=1000299.75..1036031.50 rows=714635 width=161)
3    ->  Sort  (cost=1000299.75..1002086.34 rows=714635 width=161)
       Sort Key: f23.id, f23.deptime, f23.crsdeptime, f23.arrtime, f23.crsarrtime, f11.id, f11.uniquecarrier,
4  f11.flightnum, f11.tailnum, f11.origin, f11.dest, f11.distance, f12.id, f12.uniquecarrier,
   f12.flightnum, f12.tailnum, f12.origin, f12.dest, f12.distance
5        ->  Append  (cost=303062.94..813562.76 rows=714635 width=161)
6           ->  Hash Right Join  (cost=303062.94..403208.20 rows=543073 width=161)
7              Hash Cond: (f11.id = f23.id)
8              ->  Seq Scan on flight_1_1 f11  (cost=0.00..42699.47 rows=2006447 width=56)
9              ->  Hash  (cost=287258.53..287258.53 rows=543073 width=105)
10                ->  Hash Join  (cost=38845.58..287258.53 rows=543073 width=105)
11                   Hash Cond: (f23.id = f12.id)
12                   ->  Seq Scan on flight_2_3 f23  (cost=0.00..144643.60 rows=4196699 width=48)
13                      Filter: ((deptime)::text > '1500'::text)
14                   ->  Hash  (cost=19742.93..19742.93 rows=821892 width=57)
15                      ->  Seq Scan on flight_1_2 f12  (cost=0.00..19742.93 rows=821892 width=57)
16                         Filter: ((flightnum)::text > '1000'::text)
17          ->  Hash Join  (cost=303062.94..403208.20 rows=171562 width=161)
18             Hash Cond: (f11_1.id = f23_1.id)
19             ->  Seq Scan on flight_1_1 f11_1  (cost=0.00..42699.47 rows=2006447 width=56)
20             ->  Hash  (cost=287258.53..287258.53 rows=543073 width=105)
21                ->  Hash Join  (cost=38845.58..287258.53 rows=543073 width=105)
22                   Hash Cond: (f23_1.id = f12_1.id)
23                   ->  Seq Scan on flight_2_3 f23_1  (cost=0.00..144643.60 rows=4196699 width=48)
24                      Filter: ((deptime)::text > '1500'::text)
25                   ->  Hash  (cost=19742.93..19742.93 rows=821892 width=57)
26                      ->  Seq Scan on flight_1_2 f12_1  (cost=0.00..19742.93 rows=821892 width=57)
27                         Filter: ((flightnum)::text > '1000'::text)
```

Figura 29

Anche PostgreSQL mantiene la struttura analoga a quella del confronto precedente, valgono le stesse osservazioni.

ORACLE

Operation	Object	Optimizer	Cost	Cardinality	Bytes
◢ SELECT STATEMENT		ALL_ROWS	171,597	1,681,614	287,555,994
◢ SORT (UNIQUE)			171,597	1,681,614	287,555,994
◢ UNION-ALL			0	0	0
◢ HASH JOIN (OUTER)			54,057	840,807	143,777,997
◢ HASH JOIN			37,485	840,807	93,329,577
TABLE ACCESS (FULL)	FLIGHT_1_2	ANALYZED	2,003	829,034	48,913,006
TABLE ACCESS (FULL)	FLIGHT_2_3	ANALYZED	13,584	6,306,163	327,920,476
TABLE ACCESS (FULL)	FLIGHT_1_1	ANALYZED	4,839	2,006,446	120,386,760
◢ HASH JOIN			54,303	840,807	143,777,997
◢ HASH JOIN			16,468	829,034	98,655,046
TABLE ACCESS (FULL)	FLIGHT_1_2	ANALYZED	2,003	829,034	48,913,006
TABLE ACCESS (FULL)	FLIGHT_1_1	ANALYZED	4,839	2,006,446	120,386,760
TABLE ACCESS (FULL)	FLIGHT_2_3	ANALYZED	13,584	6,306,163	327,920,476

Figura 30

Il piano di esecuzione anche in questo caso mantiene la stessa struttura già vista in precedenza, costituita ora da soli due blocchi. Come per MySQL e PostgreSQL l'unica differenza degna di nota è che sono ora presenti Hash (Inner) Join determinati dall'imposizione delle condizioni.

QUERY 4, 3 CONDIZIONI APPLICATE

SQL SERVER

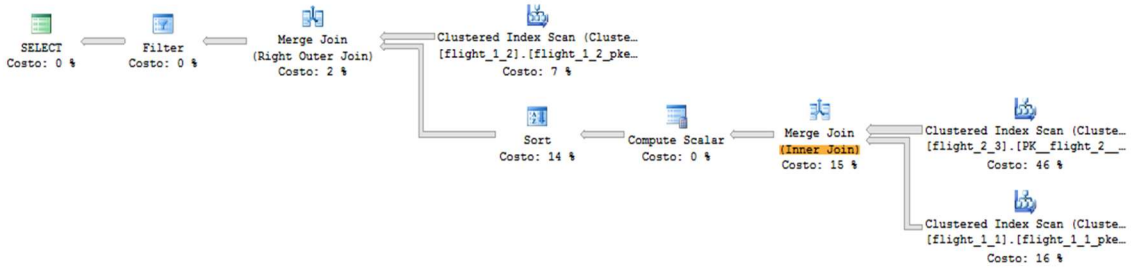


Figura 31

Il piano di esecuzione è molto simile a quello di Figura 19. La struttura è identica e l'unico aspetto differente è costituito dal fatto che il primo Merge Join eseguito realizza l'operazione logica di Inner Join, mentre nel caso precedente realizzava Left Outer Join, con flight_1_1 come left side.
Viene anche mantenuto l'operatore Filter per l'applicazione dei predicati prima della SELECT.

POSTGRESQL

```
1  QUERY PLAN
2  Hash Left Join  (cost=93560.01..379752.97 rows=12194 width=161)
3    Hash Cond: (COALESCE(f11.id, f23.id) = f12.id)
     Filter: ((((f23.deptime)::text > '1500'::text)
   AND ((f11.uniquecarrier)::text = 'EV'::text)) OR (((f11.uniquecarrier)::text = 'DL'::text)
4  AND ((f12.flightnum)::text > '1000'::text)))
5    -> Hash Join  (cost=56620.74..324301.93 rows=174643 width=104)
6         Hash Cond: (f23.id = f11.id)
7         -> Seq Scan on flight_2_3 f23  (cost=0.00..128765.28 rows=6351328 width=48)
8         -> Hash  (cost=52731.71..52731.71 rows=174643 width=56)
9              -> Seq Scan on flight_1_1 f11  (cost=0.00..52731.71 rows=174643 width=56)
10                  Filter: (((uniquecarrier)::text = 'EV'::text) OR ((uniquecarrier)::text = 'DL'::text))
11   -> Hash  (cost=17670.34..17670.34 rows=829034 width=57)
12        -> Seq Scan on flight_1_2 f12  (cost=0.00..17670.34 rows=829034 width=57)
```

Figura 32

Anche per PostgreSQL vale quanto detto per SQL Server. Il piano di esecuzione ricalca quello di Figura 20, dal quale si discosta solamente per il fatto che il primo Left Outer Join specificato nella query si trasforma in Inner Join, realizzato mediante Hash Join, per gli stessi motivi già esposti per SQL Server. Anche in questo caso, la differenza nella natura

della logica del join non porta modifiche relative alle operazioni successive nel query plan. Si sottolinea che anche in questo caso è mantenuto il filtro come penultima operazione.

ORACLE

Operation	Object	Optimizer	Cost	Cardinality	Bytes
◿ SELECT STATEMENT		ALL_ROWS	150,358	6,351,271	2,432,536,793
◿ FILTER			0	0	0
◿ HASH JOIN (RIGHT OUTER)			150,358	6,351,271	2,432,536,793
TABLE ACCESS (FULL)	FLIGHT_1_2	ANALYZED	2,002	829,034	48,913,006
◿ VIEW			44,501	6,351,271	2,057,811,804
◿ HASH JOIN (RIGHT OUTER)			44,501	6,351,271	711,342,352
TABLE ACCESS (FULL)	FLIGHT_1_1	ANALYZED	4,839	2,006,446	120,386,760
TABLE ACCESS (FULL)	FLIGHT_2_3	ANALYZED	13,575	6,351,271	330,266,092

Figura 33

Anche in questo caso il piano di esecuzione presenta la stessa struttura di quello in Figura 21.
Le uniche differenze, anche per Oracle XE, sono rappresentate dagli Hash Join, che realizzano operazioni logiche di natura differente. In Figura 21 infatti erano realizzati Full Outer Join, ricalcando quanto imposto dalla query, in quanto tutte le condizioni erano imposte appena prima della SELECT; nel caso ora considerato invece la riformulazione prevede che siano eseguiti Left Outer Join, che poi il query optimizer decide di trasformare in Right Outer Join. A differenza di SQL Server e PostgreSQL non vi è alcun join trasformato in Inner Join, in quanto Oracle XE anche in questo caso applica i 3 predicati prima della SELECT, rimanendo fedele alla scelta operata anche in merito alla Figura 21.

Dai piani ora riportati si ricava quindi che la formulazione delle query nella versione compatibile con MySQL non comporta alcun cambiamento sostanziale nel caso di applicazione di tre predicati distinti sulle tre tabelle convolte.

Si riporta anche il confronto basato sulla formulazione della query facente uso di vista intermedia.

SQL SERVER

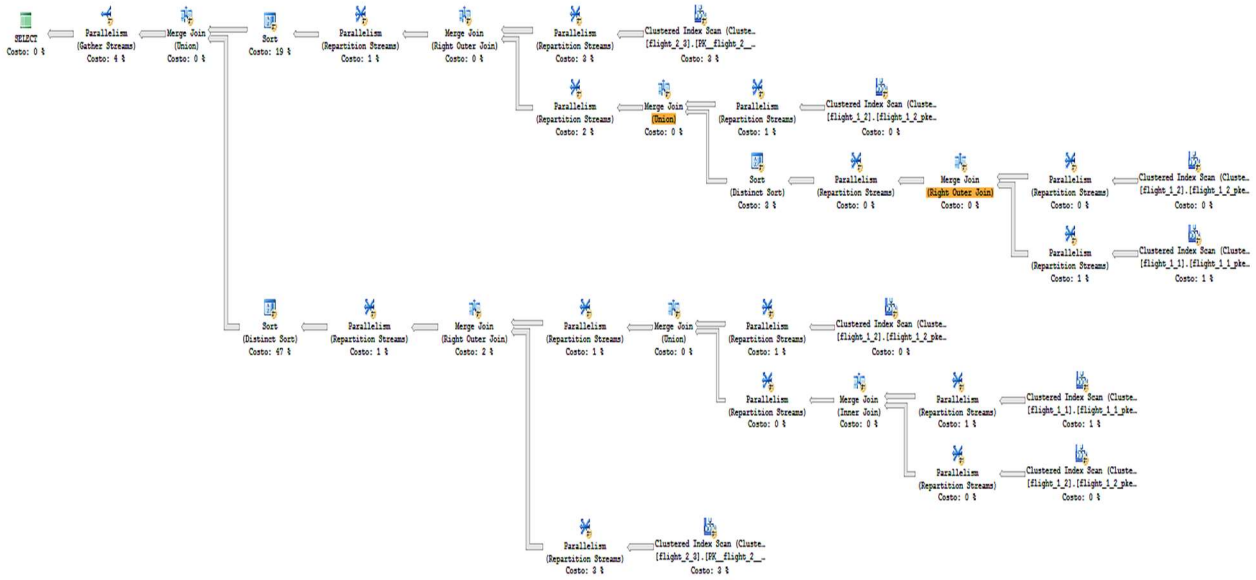


Figura 34

Rimangono valide le differenze già elencate in merito alla Figura 25.
Il piano di esecuzione presenta comunque una struttura simile a quella di MySQL: eliminando le differenze già citate, infatti, lo scheletro risulta imperniato in entrambi i casi sull'unione dei risultati provenienti da una parte dal Left Outer Join tra la vista intermedia, imputata a left side, e flight_2_3, e dall'altra dal Left Outer Join tra flight_2_3 in qualità di left side e la vista intermedia; si individuano quindi in entrambi i casi due principali blocchi speculari.
Ciò che ora cambia, e si aggiunge alle differenze tra i due piani di esecuzione, è la sequenza di operazioni che portano alla generazione della vista intermedia: MySQL per ottenerla sfrutta due Nested Loop che realizzano Left Outer Join, in cui si ha come left side dapprima flight_1_2 e poi flight_1_1, infine unisce mediante UNION i risultati ottenuti; SQL Server invece riesce a ottimizzare l'operazione e realizza prima un Right Outer Join tra flight_1_2 e flight_1_1 per poi realizzare direttamente una UNION tra il risultato del join e flight_1_2; di fatto quindi SQL Server giunge agli stessi risultati con un join in meno.
Un'ulteriore differenza è insita nel fatto che MySQL ricorre a tabelle temporanee per memorizzare i risultati della vista generata, in modo da non doverla ricreare qualora avesse bisogno di accedere nuovamente a tali dati, aspetto che torna utile nella query in esame in quanto la vista viene acceduta in entrambi i blocchi principali. SQL Server invece in questa occasione non ne fa ricorso, infatti la sua versione delle *materialized views* è rappresentata dalle *Indexed Views*, che ovviamente per essere impiegate adeguatamente devono avere specificato un indice, cosa non prevista nel caso ora considerato.

POSTGRESQL

```
1  QUERY PLAN
2  Unique  (cost=13087261.22..13385832.51 rows=9186809 width=788)
3   -> Sort  (cost=13087261.22..13110228.24 rows=9186809 width=788)
4       Sort Key: f12.id, f12.uniquecarrier, f12.flightnum, f12.tailnum, f12.origin, f12.dest, f12.distance,
f23.id, f23.deptime, f23.crsdeptime, f23.arrtime, f23.crsarrtime
5       -> Append  (cost=694931.93..2416278.25 rows=9186809 width=788)
6          -> Merge Left Join  (cost=694931.93..1162205.08 rows=2835481 width=788)
7             Merge Cond: (f12.id = f23.id)
8             -> Unique  (cost=694931.37..751640.99 rows=2835481 width=57)
9                -> Sort  (cost=694931.37..702020.08 rows=2835481 width=57)
10                  Sort Key: f12.id, f12.uniquecarrier, f12.flightnum, f12.tailnum, f12.origin, f12.dest, f12.distance
11                  -> Append  (cost=0.00..177816.40 rows=2835481 width=57)
12                     -> Seq Scan on flight_1_2 f12  (cost=0.00..17670.34 rows=829034 width=57)
13                     -> Hash Left Join  (cost=36939.26..131791.25 rows=2006447 width=57)
14                        Hash Cond: (f11.id = f12_1.id)
15                        -> Seq Scan on flight_1_1 f11  (cost=0.00..42699.47 rows=2006447 width=32)
16                        -> Hash  (cost=17670.34..17670.34 rows=829034 width=57)
17                           -> Seq Scan on flight_1_2 f12_1  (cost=0.00..17670.34 rows=829034 width=57)
18            -> Index Scan using flight_2_3_pkey on flight_2_3 f23  (cost=0.56..330887.44 rows=6351328 width=48)
19         -> Merge Right Join  (cost=694931.93..1162205.08 rows=6351328 width=788)
20            Merge Cond: (f12_2.id = f23_1.id)
21            -> Unique  (cost=694931.37..751640.99 rows=2835481 width=57)
22               -> Sort  (cost=694931.37..702020.08 rows=2835481 width=57)
23                  Sort Key: f12_2.id, f12_2.uniquecarrier, f12_2.flightnum, f12_2.tailnum, f12_2.origin, f12_2.dest, f12_2.distance
24                  -> Append  (cost=0.00..177816.40 rows=2835481 width=57)
25                     -> Seq Scan on flight_1_2 f12_2  (cost=0.00..17670.34 rows=829034 width=57)
26                     -> Hash Left Join  (cost=36939.26..131791.25 rows=2006447 width=57)
27                        Hash Cond: (f11_1.id = f12_3.id)
28                        -> Seq Scan on flight_1_1 f11_1  (cost=0.00..42699.47 rows=2006447 width=32)
29                        -> Hash  (cost=17670.34..17670.34 rows=829034 width=57)
30                           -> Seq Scan on flight_1_2 f12_3  (cost=0.00..17670.34 rows=829034 width=57)
31            -> Index Scan using flight_2_3_pkey on flight_2_3 f23_1  (cost=0.56..330887.44 rows=6351328 width=48)
```

Figura 35

Anche per PostgreSQL possono essere tratte le stesse conclusioni a cui si è giunti per SQL Server: trascurando le differenze rispetto il piano prodotto da MySQL, già esposte e che rimangono valide anche in questo caso, le uniche due differenze ulteriori che si riscontrano sono: la maggiore semplicità nelle operazioni che portano alla generazione della vista intermedia, costituite da un unico join tra flight_1_1 e flight_1_2, seguito dalle operazioni adottate da PostgreSQL per la realizzazione della UNION (Append, Sort, Unique); la mancanza da parte di PostgreSQL dell'adozione di una tabella allo scopo di memorizzare temporaneamente i risultati della vista, in modo da accelerare l'esecuzione della query, non dovendo ricostruirla, come invece accade.
Si noti che anche in questo caso PostgreSQL ricorre a Merge Join, in modo da facilitare le successive operazioni di sort.

ORACLE

Operation	Object	Optimizer	Cost	Cardinality	Bytes
SELECT STATEMENT		ALL_ROWS	741,087	9,228,460	2,224,058,860
SORT (UNIQUE)			741,087	9,228,460	2,224,058,860
UNION-ALL			0	0	0
HASH JOIN (RIGHT OUTER)			129,928	2,877,189	693,402,549
TABLE ACCESS (FULL)	FLIGHT_2_3	ANALYZED	13,575	6,351,271	330,266,092
VIEW	RESPARTIALJOIN		70,156	2,835,480	535,905,720
SORT (UNIQUE)			70,156	2,835,480	233,506,038
UNION-ALL			0	0	0
TABLE ACCESS (FULL)	FLIGHT_1_2	ANALYZED	2,002	829,034	48,913,006
HASH JOIN (RIGHT OUTER)			13,900	2,006,446	184,593,032
TABLE ACCESS (FULL)	FLIGHT_1_2	ANALYZED	2,002	829,034	48,913,006
TABLE ACCESS (FULL)	FLIGHT_1_1	ANALYZED	4,832	2,006,446	66,212,718
HASH JOIN (OUTER)			129,938	6,351,271	1,530,656,311
TABLE ACCESS (FULL)	FLIGHT_2_3	ANALYZED	13,575	6,351,271	330,266,092
VIEW	RESPARTIALJOIN		70,156	2,835,480	535,905,720
SORT (UNIQUE)			70,156	2,835,480	233,506,038
UNION-ALL			0	0	0
TABLE ACCESS (FULL)	FLIGHT_1_2	ANALYZED	2,002	829,034	48,913,006
HASH JOIN (RIGHT OUTER)			13,900	2,006,446	184,593,032
TABLE ACCESS (FULL)	FLIGHT_1_2	ANALYZED	2,002	829,034	48,913,006
TABLE ACCESS (FULL)	FLIGHT_1_1	ANALYZED	4,832	2,006,446	66,212,718

Figura 36

Rimangono valide le differenze già citate in precedenza. Anche in questo caso il piano di esecuzione presenta una struttura analoga a quella di MySQL, e vi si può riscontrare come unica differenza, che si somma alle precedenti, quella individuata anche per SQL Server e PostgreSQL: la generazione della vista anche in Oracle XE passa attraverso solo un Right Outer Join, il cui risultato è poi unito alla tabella rimanente solo mediante le operazioni che realizzano la UNION (Union-all e Sort unique), a differenza di MySQL che realizza due Left Outer Join e una successiva Union.
Anche Oracle non fa ricorso a tabelle temporanee (materialized views) che potrebbero ridurre il tempo impiegato per portare a termine l'esecuzione della query.

7.1.2 Gruppo 2

7.1.2.1 Query 1, nessuna condizione imposta

```
SELECT *
FROM flight_1_1 f11
FULL OUTER JOIN flight_1_2 f12 ON f11.id=f12.id
FULL OUTER JOIN flight_2_1 f21 ON f21.id=COALESCE(f11.id, f12.id)
```

Versione per MySQL:

```
SELECT f21. * , f11. * , f12. *
FROM flight_2_1 f21
LEFT OUTER JOIN flight_1_1 f11 ON f11.id=f21.id
LEFT OUTER JOIN flight_1_2 f12 ON f12.id=f21.id
UNION
SELECT f21. * , f11. * , f12. *
FROM flight_1_1 f11
LEFT OUTER JOIN flight_2_1 f21 ON f21.id=f11.id
LEFT OUTER JOIN flight_1_2 f12 ON f12.id=f11.id
UNION
SELECT f21. * , f11. * , f12. *
FROM flight_1_2 f12
LEFT OUTER JOIN flight_2_1 f21 ON f12.id=f21.id
LEFT OUTER JOIN flight_1_1 f11 ON f11.id=f12.id
```

- *Righe ritornate:* ***2,901,770***
- *Tempi di esecuzione:*

MS SQL Server	*PostgreSQL*	*Oracle*	*MySQL*
00:00:36	*00:04:58*	*01:17:00*	*25:34:00*

- *Piani di esecuzione*
  MS SQL Server:

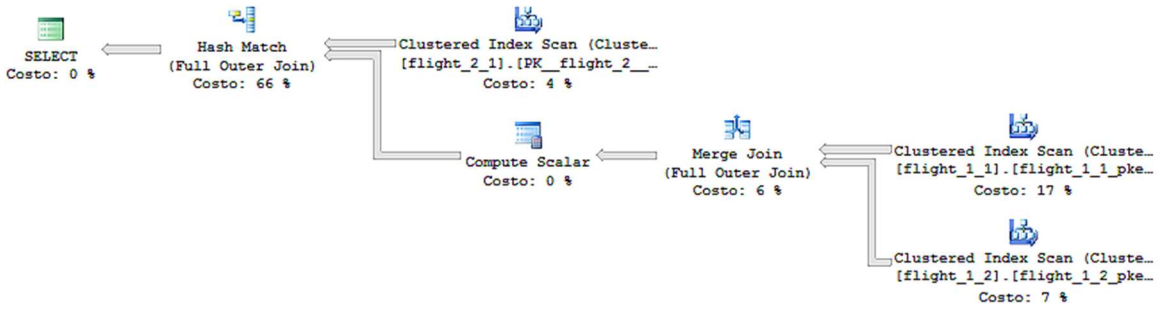


Figura 37

PostgreSQL:

1	QUERY PLAN
2	Hash Full Join (cost=57693.74..266825.38 rows=2006447 width=161)
3	Hash Cond: (COALESCE(f11.id, f12.id) = f21.id)
4	-> Hash Full Join (cost=36939.26..143549.25 rows=2006447 width=113)
5	Hash Cond: (f11.id = f12.id)
6	-> Seq Scan on flight_1_1 f11 (cost=0.00..42699.47 rows=2006447 width=56)
7	-> Hash (cost=17670.34..17670.34 rows=829034 width=57)
8	-> Seq Scan on flight_1_2 f12 (cost=0.00..17670.34 rows=829034 width=57)
9	-> Hash (cost=10113.99..10113.99 rows=499799 width=48)
10	-> Seq Scan on flight_2_1 f21 (cost=0.00..10113.99 rows=499799 width=48)

Figura 38

Oracle XE:

Operation	Object	Optimizer	Cost	Cardinality	Bytes
SELECT STATEMENT		ALL_ROWS	56,042	4,526,881	2,322,289,953
VIEW	VW_FOJ_0		56,042	4,526,881	2,322,289,953
HASH JOIN (FULL OUTER)			56,042	4,526,881	1,946,558,830
TABLE ACCESS (FULL)	FLIGHT_2_1	ANALYZED	1,054	499,798	25,989,496
VIEW	VW_FOJ_1		16,467	2,006,446	758,436,588
HASH JOIN (FULL OUTER)			16,467	2,006,446	238,767,074
TABLE ACCESS (FULL)	FLIGHT_1_2	ANALYZED	2,002	829,034	48,913,006
TABLE ACCESS (FULL)	FLIGHT_1_1	ANALYZED	4,839	2,006,446	120,386,760

Figura 39

MySQL:

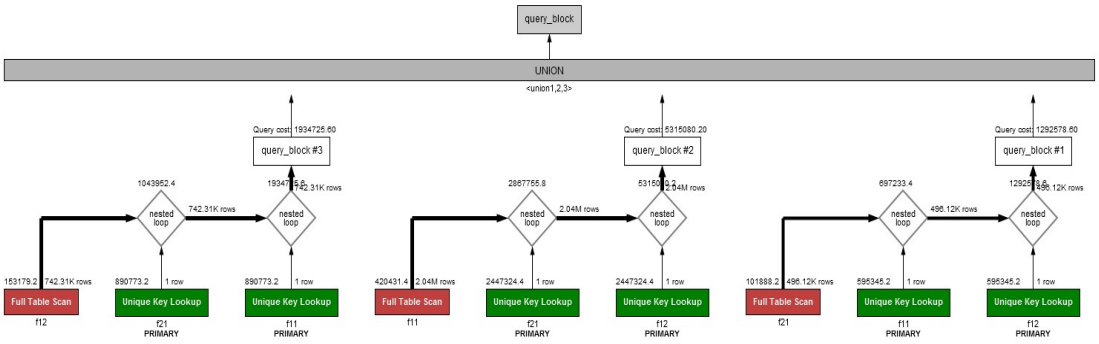


Figura 40

7.1.2.2 Query 2, singola condizione, poco restrittiva

```
SELECT *
FROM flight_1_1 f11
FULL OUTER JOIN flight_1_2 f12 ON f11.id=f12.id
FULL OUTER JOIN flight_2_1 f21 ON f21.id=COALESCE(f11.id, f12.id)
WHERE f11.distance > '100'
```

Versione per MySQL:

```
SELECT f21. * , f11. * , f12. *
FROM flight_2_1 f21
LEFT OUTER JOIN flight_1_1 f11 ON f11.id=f21.id
LEFT OUTER JOIN flight_1_2 f12 ON f12.id=f21.id
WHERE f11.distance > '100'
UNION
SELECT f21. * , f11. * , f12. *
FROM flight_1_1 f11
LEFT OUTER JOIN flight_2_1 f21 ON f21.id=f11.id
LEFT OUTER JOIN flight_1_2 f12 ON f12.id=f11.id
WHERE f11.distance > '100'
UNION
SELECT f21. * , f11. * , f12. *
FROM flight_1_2 f12
LEFT OUTER JOIN flight_2_1 f21 ON f12.id=f21.id
LEFT OUTER JOIN flight_1_1 f11 ON f11.id=f12.id
WHERE f11.distance > '100'
```

- *Righe ritornate:* ***1,982,047***
- *Tempi di esecuzione:*

MS SQL Server	*PostgreSQL*	*Oracle*	*MySQL*
00:00:27	*00:03:17*	*00:15:33*	*00:25:51*

- *Piani di esecuzione*

  MS SQL Server:

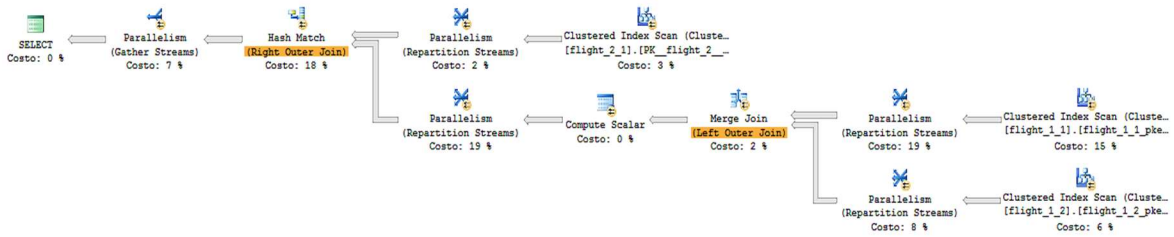


Figura 41

PostgreSQL:

```
1  QUERY PLAN
2  Hash Left Join (cost=57693.74..269885.18 rows=1980822 width=161)
3    Hash Cond: (COALESCE(f11.id, f12.id) = f21.id)
4    -> Hash Left Join (cost=36939.26..147861.40 rows=1980822 width=113)
5         Hash Cond: (f11.id = f12.id)
6         -> Seq Scan on flight_1_1 f11  (cost=0.00..47715.59 rows=1980822 width=56)
7              Filter: ((distance)::text > '100'::text)
8         -> Hash  (cost=17670.34..17670.34 rows=829034 width=57)
9              -> Seq Scan on flight_1_2 f12  (cost=0.00..17670.34 rows=829034 width=57)
10   -> Hash  (cost=10113.99..10113.99 rows=499799 width=48)
11        -> Seq Scan on flight_2_1 f21  (cost=0.00..10113.99 rows=499799 width=48)
```

Figura 42

Oracle XE:

Operation	Object	Optimizer	Cost	Cardinality	Bytes
◢ SELECT STATEMENT		ALL_ROWS	55,884	2,013,571	865,835,530
◢ HASH JOIN (RIGHT OUTER)			55,884	2,013,571	865,835,530
TABLE ACCESS (FULL)	FLIGHT_2_1	ANALYZED	1,054	499,798	25,989,496
◢ VIEW	VW_FOJ_0		16,446	1,999,430	755,784,540
◢ HASH JOIN (RIGHT OUTER)			16,446	1,999,430	237,932,170
TABLE ACCESS (FULL)	FLIGHT_1_2	ANALYZED	2,002	829,034	48,913,006
TABLE ACCESS (FULL)	FLIGHT_1_1	ANALYZED	4,841	1,999,430	119,965,800

Figura 43

MySQL:

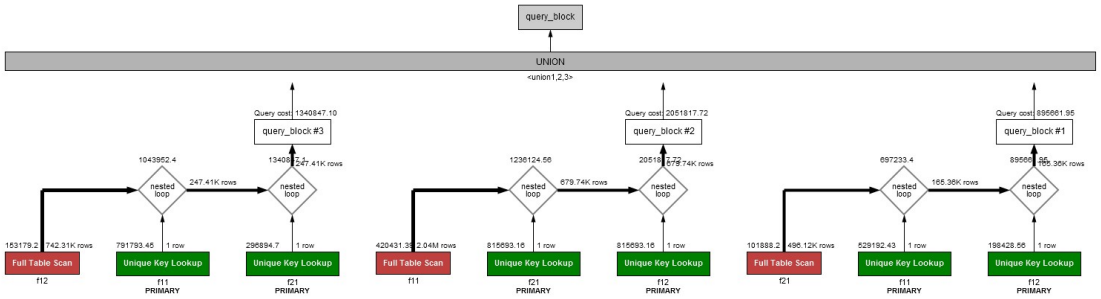


Figura 44

7.1.2.3 Query 3, due condizioni imposte, poco selettive

```
SELECT *
FROM flight_1_1 f11
FULL OUTER JOIN flight_1_2 f12 ON f11.id=f12.id
FULL OUTER JOIN flight_2_1 f21 ON f21.id=COALESCE(f11.id, f12.id)
WHERE f11.distance > '100'
AND f12.uniquecarrier <> 'YV'
```

Versione per MySQL:

```
SELECT f21. * , f11. * , f12. *
FROM flight_2_1 f21
LEFT OUTER JOIN flight_1_1 f11 ON f11.id=f21.id
LEFT OUTER JOIN flight_1_2 f12 ON f12.id=f21.id
WHERE f11.distance > '100'
AND f12.uniquecarrier <> 'YV'
UNION
SELECT f21. * , f11. * , f12. *
FROM flight_1_1 f11
LEFT OUTER JOIN flight_2_1 f21 ON f21.id=f11.id
LEFT OUTER JOIN flight_1_2 f12 ON f12.id=f11.id
WHERE f11.distance > '100'
AND f12.uniquecarrier <> 'YV'
```

- *Righe ritornate:* ***184,359***
- *Tempi di esecuzione:*

MS SQL Server	*PostgreSQL*	*Oracle*	*MySQL*
00:00:02.8	*00:00:16.7*	*00:01:24*	*00:02:01*

- *Piani di esecuzione*

  MS SQL Server:

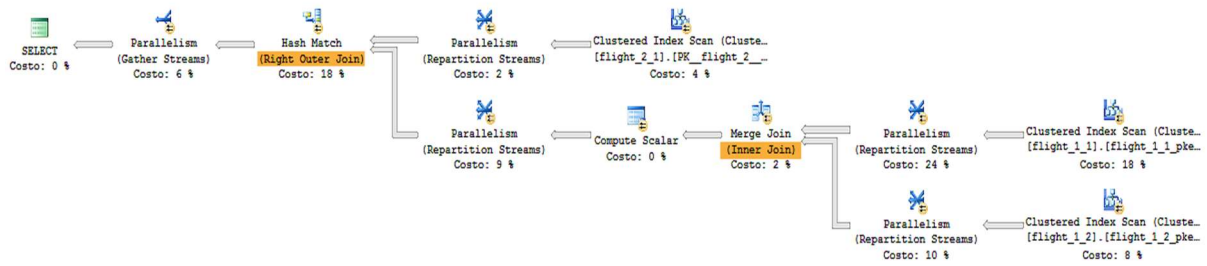


Figura 45

PostgreSQL:

1	QUERY PLAN
2	Hash Left Join (cost=59651.84..214656.27 rows=813590 width=161)
3	Hash Cond: (COALESCE(f11.id, f12.id) = f21.id)
4	-> Hash Join (cost=38897.36..149717.93 rows=813590 width=113)
5	Hash Cond: (f11.id = f12.id)
6	-> Seq Scan on flight_1_1 f11 (cost=0.00..47715.59 rows=1980822 width=56)
7	Filter: ((distance)::text > '100'::text)
8	-> Hash (cost=19742.93..19742.93 rows=824115 width=57)
9	-> Seq Scan on flight_1_2 f12 (cost=0.00..19742.93 rows=824115 width=57)
10	Filter: ((uniquecarrier)::text <> 'YV'::text)
11	-> Hash (cost=10113.99..10113.99 rows=499799 width=48)
12	-> Seq Scan on flight_2_1 f21 (cost=0.00..10113.99 rows=499799 width=48)

Figura 46

Oracle XE:

Operation	Object	Optimizer	Cost	Cardinality	Bytes
SELECT STATEMENT		ALL_ROWS	33,099	779,237	335,071,910
HASH JOIN (RIGHT OUTER)			33,099	779,237	335,071,910
TABLE ACCESS (FULL)	FLIGHT_2_1	ANALYZED	1,054	499,798	25,989,496
VIEW	VW_FOJ_0		16,262	773,765	292,483,170
HASH JOIN			16,262	773,765	92,078,035
TABLE ACCESS (FULL)	FLIGHT_1_2	ANALYZED	2,003	773,765	45,652,135
TABLE ACCESS (FULL)	FLIGHT_1_1	ANALYZED	4,841	1,999,430	119,965,800

Figura 47

Si sottolinea come anche nel piano ora riportato le condizioni, entrambe poco selettive, siano applicate tramite gli operatori Table Access (Full) sulle due relazioni interessate. Lo si può determinare dalla figura sopra riportata, che registra una cardinalità per flight_1_1 e flight_1_2 ridotte rispetto il numero totale di righe contenute nelle due tabelle.

MySQL:

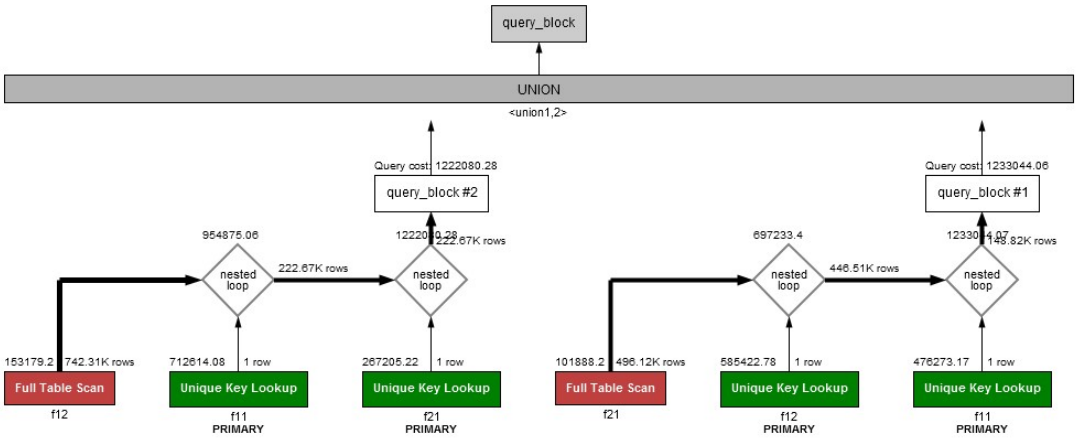


Figura 48

7.1.2.4 Query 4, due condizioni imposte, selettive

```
SELECT *
FROM flight_1_1 f11
FULL OUTER JOIN flight_1_2 f12 ON f11.id=f12.id
FULL OUTER JOIN flight_2_1 f21 ON f21.id=COALESCE(f11.id, f12.id)
WHERE f11.uniquecarrier='EV' OR f11.uniquecarrier='DL'
AND f12.uniquecarrier = 'YV'
```

Versione per MySQL:

```
SELECT f21. * , f11. * , f12. *
FROM flight_2_1 f21
LEFT OUTER JOIN flight_1_1 f11 ON f11.id=f21.id
LEFT OUTER JOIN flight_1_2 f12 ON f12.id=f21.id
WHERE f11.uniquecarrier='EV' OR f11.uniquecarrier='DL'
AND f12.uniquecarrier = 'YV'
UNION
SELECT f21. * , f11. * , f12. *
FROM flight_1_1 f11
LEFT OUTER JOIN flight_2_1 f21 ON f21.id=f11.id
LEFT OUTER JOIN flight_1_2 f12 ON f12.id=f11.id
WHERE f11.uniquecarrier='EV' OR f11.uniquecarrier='DL'
AND f12.uniquecarrier = 'YV'
```

- *Righe ritornate:* ***108,379***
- *Tempi di esecuzione:*

MS SQL Server	*PostgreSQL*	*Oracle*	*MySQL*
00:00:02.3	*00:00:16.7*	*00:02:54*	*00:01:10*

- *Piani di esecuzione*

  MS SQL Server:

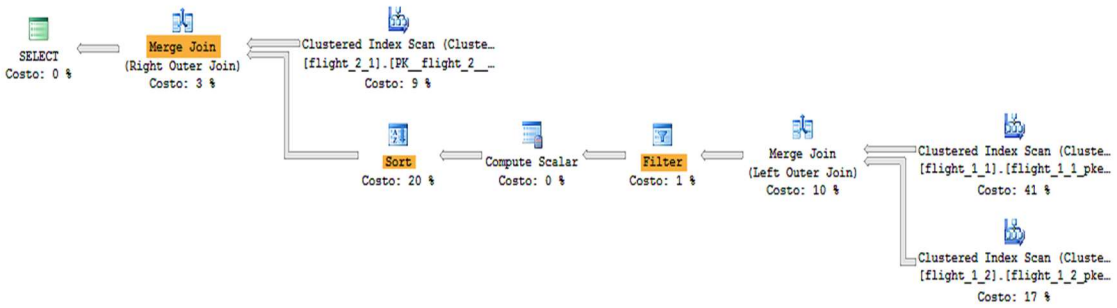


Figura 49

PostgreSQL:

```
QUERY PLAN
Nested Loop Left Join (cost=56621.17..104020.90 rows=9077 width=161) (actual time=2146.850..14073.919 rows=108379 loops=1)
  -> Hash Right Join  (cost=56620.74..98180.76 rows=9077 width=113) (actual time=2146.802..10700.747 rows=108379 loops=1)
      Hash Cond: (f12.id = f11.id)
      Filter: (((f11.uniquecarrier)::text = 'EV'::text) OR (((f11.uniquecarrier)::text = 'DL'::text) AND ((f12.uniquecarrier)::text = 'YV'::text)))
      Rows Removed by Filter: 71978
      -> Seq Scan on flight_1_2 f12  (cost=0.00..17670.34 rows=829034 width=57) (actual time=0.049..3737.870 rows=829034 loops=1)
      -> Hash  (cost=52731.71..52731.71 rows=174643 width=56) (actual time=2136.612..2136.612 rows=180357 loops=1)
          Buckets: 65536  Batches: 8  Memory Usage: 2540kB
          -> Seq Scan on flight_1_1 f11  (cost=0.00..52731.71 rows=174643 width=56) (actual time=0.331..1269.364 rows=180357 loops=1)
              Filter: (((uniquecarrier)::text = 'EV'::text) OR ((uniquecarrier)::text = 'DL'::text))
              Rows Removed by Filter: 1826090
  -> Index Scan using flight_2_1_pkey on flight_2_1 f21  (cost=0.42..0.63 rows=1 width=48) (actual time=0.015..0.016 rows=0 loops=108379)
      Index Cond: (id = COALESCE(f11.id, f12.id))
```

Figura 50

Oracle XE:

1 - filter(("F11"."UNIQUECARRIER"='EV' OR ("F11"."UNIQUECARRIER"='DL' AND "F12"."UNIQUECARRIER"='YV')))

Operation	Object	Optimizer	Cost	Cardinality	Bytes
SELECT STATEMENT		ALL_ROWS	56,042	4,526,881	2,322,289,953
VIEW	VW_FOJ_0		56,042	4,526,881	2,322,289,953
HASH JOIN (FULL OUTER)			56,042	4,526,881	1,946,558,830
TABLE ACCESS (FULL)	FLIGHT_2_1	ANALYZED	1,054	499,798	25,989,496
VIEW	VW_FOJ_1		16,467	2,006,446	758,436,588
HASH JOIN (FULL OUTER)			16,467	2,006,446	238,767,074
TABLE ACCESS (FULL)	FLIGHT_1_2	ANALYZED	2,002	829,034	48,913,006
TABLE ACCESS (FULL)	FLIGHT_1_1	ANALYZED	4,839	2,006,446	120,386,760

Figura 51

MySQL:

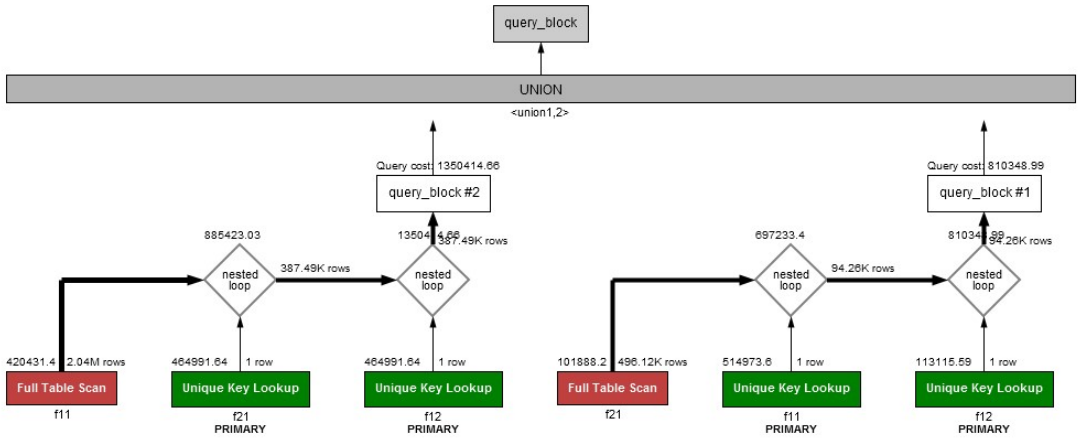


Figura 52

7.1.2.5 Query 5, tre condizioni imposte, poco selettive

```
SELECT *
FROM flight_1_1 f11
FULL OUTER JOIN flight_1_2 f12 ON f11.id=f12.id
FULL OUTER JOIN flight_2_1 f21 ON f21.id=COALESCE(f11.id, f12.id)
WHERE f11.distance > '100'
AND f12.uniquecarrier <> 'YV'
AND f21.deptime <> 'NA'
f12.uniquecarrier = 'YV'
```

Versione per MySQL:

```
SELECT *
FROM flight_1_1 f11
LEFT OUTER JOIN flight_1_2 f12 ON f11.id=f12.id
LEFT OUTER JOIN flight_2_1 f21 ON f21.id=COALESCE(f11.id, f12.id)
WHERE f11.distance > '100'
AND f12.uniquecarrier <> 'YV'
AND f21.deptime <> 'NA'
```

- *Righe ritornate:* ***20,086***
- *Tempi di esecuzione:*

MS SQL Server	*PostgreSQL*	*Oracle*	*MySQL*
00:00:01	*00:00:05.4*	*00:01:13*	*00:00:12*

- *Piani di esecuzione*

  MS SQL Server:

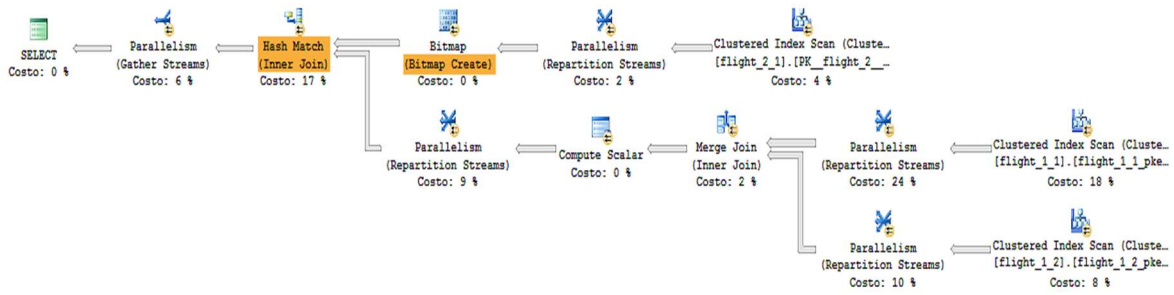


Figura 53

PostgreSQL:

```
1  QUERY PLAN
2  Hash Join  (cost=60718.88..215508.37 rows=799596 width=161)
3    Hash Cond: (COALESCE(f11.id, f12.id) = f21.id)
4    ->  Hash Join  (cost=38897.36..149717.93 rows=813590 width=113)
5          Hash Cond: (f11.id = f12.id)
6          ->  Seq Scan on flight_1_1 f11  (cost=0.00..47715.59 rows=1980822 width=56)
7                Filter: ((distance)::text > '100'::text)
8          ->  Hash  (cost=19742.93..19742.93 rows=824115 width=57)
9                ->  Seq Scan on flight_1_2 f12  (cost=0.00..19742.93 rows=824115 width=57)
10                     Filter: ((uniquecarrier)::text <> 'YV'::text)
11   ->  Hash  (cost=11363.49..11363.49 rows=491202 width=48)
12         ->  Seq Scan on flight_2_1 f21  (cost=0.00..11363.49 rows=491202 width=48)
13               Filter: ((deptime)::text <> 'NA'::text)
```

Figura 54

Oracle XE:

filter("F21"."DEPTIME"<>'NA')

Operation	Object	Optimizer	Cost	Cardinality	Bytes
SELECT STATEMENT		ALL_ROWS	33,099	778,582	334,790,260
HASH JOIN			33,099	778,582	334,790,260
TABLE ACCESS (FULL)	FLIGHT_2_1	ANALYZED	1,055	499,378	25,967,656
VIEW	VW_FOJ_0		16,262	773,765	292,483,170
HASH JOIN			16,262	773,765	92,078,035
TABLE ACCESS (FULL)	FLIGHT_1_2	ANALYZED	2,003	773,765	45,652,135
TABLE ACCESS (FULL)	FLIGHT_1_1	ANALYZED	4,841	1,999,430	119,965,800

filter("F11"."DISTANCE">'100')

filter("F12"."UNIQUECARRIER"<>'YV')

Figura 55

MySQL:

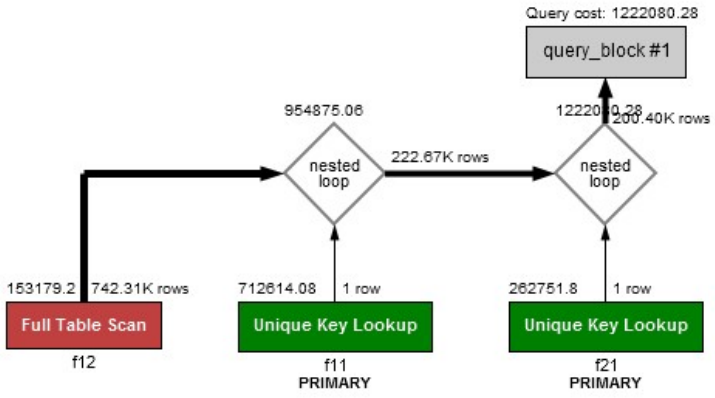


Figura 56

7.1.2.6 Comparazione tra MS SQL Server, PostgreSQL e Oracle XE.

Anche per il secondo gruppo, per SQL Server e Oracle XE sono state eseguite alcune query sia nella versione che prevede l'utilizzo di COALESCE, sia senza; rimangono valide le osservazioni effettuate per il primo gruppo, senza riscontrare alcuna differenza per SQL Server, mentre per quanto riguarda Oracle XE si è notato un importante aspetto: quando l'individuazione a tempo di esecuzione delle sole righe effettivamente coinvolte nel match, per le quali sono verificate le condizioni di join, avviene su una tabella con cardinalità non elevata - nel caso in esame 499,799 righe - viene preferito applicare un'operazione di Full Table Access, piuttosto che realizzare l'index seek dinamico presentato in relazione al gruppo 1.

Quindi si può concludere che, per elaborare una condizione di join complessa perché basata sull'unione mediante OR di due predicati, sia SQL Server che Oracle XE implementano e realizzano un Index Seek dinamico, per SQL Server basato su range non sovrapposti (Merge Interval), per Oracle XE basato sulla ricerca di una singola riga acceduta mediante Index Unique Scan.

Inoltre SQL Server applica la medesima strategia anche in relazione a tabelle di non elevate dimensioni, mentre Oracle XE invece per tabelle che contano numerose righe - dal gruppo 1, circa 900 mila - applica la strategia appena esposta, sfruttando l'Index Seek dinamico, mentre se le dimensioni delle tabelle sono inferiori - circa 500 mila dal gruppo corrente -, non accede direttamente ad una specifica riga, ma scorre per intero la tabella per trovare la riga con la quale esiste il match, perché risulta essere un'operazione meno onerosa.

In generale, sia per SQL Server sia per PostgreSQL e Oracle XE, possono essere tratte le stesse conclusioni del gruppo 1, che trovano quindi conferma anche in questo secondo gruppo.

In particolare per SQL Server si riconfermano le motivazioni che spingono all'adozione del parallelismo, individuate precedentemente; quindi, ancora una volta, si può affermare che SQL Server decida di prediligere il piano parallelo dato l'aumento del costo associato, determinato dall'inversione della relazione di dimensioni maggiori con quella di dimensioni minori, che costringe l'Hash Match a dover processare un numero più elevato di righe e a generare in memoria una hash table (operazione onerosa) a partire da una relazione più grande.

In merito alla Figura 41 e alla Figura 45 si riconferma invece che SQL Server opta per il piano parallelo anche nel caso in cui vengano imposte delle condizioni su tabelle di dimensioni considerevoli e i predicati impongono quindi all'operazione di Clustered Index Scan di scorrere ed in più analizzare ogni singola riga, portando ad un complessivo incremento di CPU time che diventa significativo per il costo dell'operatore stesso.

Inoltre i dati ricavati in merito alla Figura 41 confermano e mettono maggiormente in evidenza che il costo del piano seriale della query in cui non vengono imposte condizioni (Figura 37) è inferiore rispetto quello parallelo perché il parallelismo porterebbe in effetti ad un calo considerevole dei costi dei due join ma tale calo sarebbe inferiore rispetto ai costi aggiuntivi degli Exchange operators. Infatti dai dati citati si ha che per la query di Figura 41 il costo del piano parallelo è minore di quello seriale di appena 0.8269, e questo quindi dimostra che il beneficio del parallelismo subentra poi quando il piano seriale subisce un aumento di costo dovuto all'obbligo di analizzare le righe di una tabella di elevate dimensioni per l'applicazione di un filtro; il

parallelismo, diminuendo tale costo aggiuntivo, risulta quindi lievemente vantaggioso, mentre nel caso in cui non vengano applicate condizioni è più vantaggioso il piano seriale.
Rimanendo comunque valido ciò che era stato affermato nelle considerazioni del gruppo 1, cioè che spesso il piano seriale è più oneroso dell'analogo parallelo ma può essere scelto ugualmente a causa di statistiche non aggiornate, si può invece affermare che negli esempi riportati (Figura 6 e Figura 37) è stato privilegiato quello seriale perché si trattava effettivamente del meno costoso per i motivi detti.
Per comprendere meglio il peso rivestito dagli Exchange Operators, si può sfruttare il paragone introdotto da Brent Ozar [8]: si può immaginare che una query sia un compito assegnato ad una classe di studenti, che a loro volta rappresentano i core della CPU; tale compito consiste ad esempio nel costruire nel minor tempo possibile una lista di tutte le parole distinte contenute in un libro e la relativa frequenza con cui compaiono.
Per realizzare il progetto la maestra, che rappresenta il thread master, può decidere di:

- affidare l'intero libro ad un solo studente, che ha quindi l'onere di portare a termine il progetto da solo; ovviamente così facendo sarà necessario un tempo piuttosto lungo per arrivare al completamento del compito (*esecuzione seriale*);
- suddividere le pagine del libro e assegnarne un gruppo ad ogni studente o ad un sottoinsieme di questi; in questo modo gli studenti lavorano contemporaneamente, riducendo il tempo necessario per completare il compito; si aggiunge però un tempo extra necessario per coordinare il lavoro degli studenti e unire il lavoro di ognuno in un'unica lista finale contenente i risultati, associato agli Exchange Operators (*esecuzione parallela*).

Continuando il paragone, si può dire che, se il compito assegnato non si presta particolarmente bene ad essere eseguito da più studenti, assegnarlo comunque al gruppo e non ad un solo studente fa sì ovviamente che il lavoro simultaneo riduca il tempo impiegato per l'esecuzione del progetto in sé, a questo tempo va però aggiunto quello necessario per il coordinamento degli studenti e quello per unirne i risultati, tempi che possono essere anche piuttosto elevati soprattutto nel caso in cui ogni studente debba restituire un gran quantitativo di dati, che devono essere coerenti e amalgamati tra loro (*Gather Streams*).
Come si può notare dai piani di esecuzione, quando vengono imposte delle condizioni poco restrittive SQL Server si affida alla tecnica del parallelismo, per i motivi già esaminati; in queste circostanze il "compito" affidato si presta maggiormente ad essere eseguito in parallelo rispetto alla query in cui non sono imposte condizioni. Rimangono sempre elevati i costi dovuti alle operazioni per gestire il parallelismo perché ogni core deve analizzare e ritornare sempre lo stesso numero considerevole di righe, ma la presenza dei predicati applicati con le operazioni di Clustered Index Scan, impone un ulteriore "compito" che viene eseguito in minor tempo se spartito tra più "studenti".

Dalla Figura 49 si nota un caso in cui il filtro bitmap è utilizzato in relazione ad un Merge Join, dimostrando appunto che, anche se più frequenti i casi in cui accompagna un Hash Match, può essere utilizzato anche con questa tipologia di join.

Un altro aspetto riconfermato è che sia PostgreSQL sia SQL Server e Oracle XE, in presenza di una query con condizioni tutte poco selettive, che siano 2 o 3, operano applicandole direttamente in concomitanza all'operatore Clustered Index Scan (SQL Server)/Sequence Scan (PostgreSQL)/Table Access (Oracle), senza ricorrere ad un operatore di filtraggio successivo.
Sempre per quanto riguarda i filtri, nel gruppo 1 era stato dimostrato che in presenza di una condizione poco restrittiva e una molto restrittiva, SQL Server e PostgreSQL applicavano entrambi solo quella molto selettiva per effettuare poi secondariamente anche il secondo filtro. In questo gruppo sono state applicate entrambe le condizioni molto selettive, imponendole sulle due tabelle appartenenti a sottoalberi diversi. Si può affermare che SQL Server e PostgreSQL mantengono il comportamento adottato per il gruppo 1, quindi applicano subito solo la condizione più restrittiva e la seconda la applicano successivamente, in modo da sfruttare il fatto che molte righe sono state scartate dal primo join e il secondo filtro può quindi processare un numero ridotto di righe.
Sempre in merito ai filtri, anche Oracle XE, mantiene la strategia già adottata nel gruppo 1, cioè, in presenza di almeno una condizione selettiva, le applica tutte appena prima della SELECT, registrando un comportamento differente rispetto quello di SQL Server e PostgreSQL, con gli svantaggi già illustrati. Tale strategia comporta per Oracle XE tempi di esecuzione maggiori in corrispondenza di condizioni selettive, mentre per gli altri DBMS, filtri più restrittivi equivalgono sempre a tempi di esecuzione inferiori.

Un'ultima osservazione è che PostgreSQL in Figura 50 utilizza un Nested Loop Left Join indicizzato; quindi a differenza di SQL Server non esegue un Merge Join, per i motivi già descritti al gruppo 1, ma comunque sfrutta la situazione e abbandona un più oneroso Hash Left Join per un join reso più favorevole dai seguenti fattori: colonne di join indicizzate, outer input di dimensioni ristrette e poche righe che soddisfano la condizione di join.
Al contrario, Oracle XE, nonostante in Figura 51 si trovi nella stessa situazione, continua a realizzare il join sfruttando l'algoritmo di Hash join, sebbene rimangano validi anche nel suo caso i requisiti che rendono i Nested Loop indicizzati più performanti rispetto gli altri due algoritmi di join.

7.1.2.7 Comparazione con MySQL

Rimangono valide le osservazioni fatte per il gruppo 1.
I piani di esecuzione mantengono la stessa struttura già esaminata nel gruppo 1, senza apportare modifiche rilevanti.
Si riportano per completezza nel confronto tutti i tempi di esecuzione nella tabella conclusiva.

7.1.3 Gruppo 3

Si riportano solo alcuni piani di esecuzione relativi a MS SQL Server, in quanto per PostgreSQL, Oracle XE e MySQL, il query optimizer produce dei piani con la medesima struttura di quelli riportati in relazione ai primi due gruppi, senza elementi differenti da segnalare.
I piani riportati per MS SQL Server sono solamente quelli che introducono elementi differenti rispetto a quanto precedentemente analizzato.

7.1.3.1 Query 1, nessuna condizione imposta

```
SELECT *
FROM flight_1_2 f12
FULL OUTER JOIN flight_2_1 f21 ON f21.id=f12.id
FULL OUTER JOIN flight_3_2 f32 ON f32.id=COALESCE(f12.id, f21.id)
```

Versione per MySQL:

```
SELECT f21. * , f32. * , f12. *
FROM flight_1_2 f12
LEFT OUTER JOIN flight_2_1 f21 ON f21.id=f12.id
LEFT OUTER JOIN flight_3_2 f32 ON f32.id=f12.id
UNION
SELECT f21. * , f32. * , f12. *
FROM flight_2_1 f21
LEFT OUTER JOIN flight_1_2 f12 ON f21.id=f12.id
LEFT OUTER JOIN flight_3_2 f32 ON f32.id=f21.id
UNION
SELECT f21. * , f32. * , f12. *
FROM flight_3_2 f32
LEFT OUTER JOIN flight_2_1 f21 ON f32.id=f21.id
LEFT OUTER JOIN flight_1_2 f12 ON f12.id=f32.id
```

- *Righe ritornate:* ***1,574,218***
- *Tempi di esecuzione:*

MS SQL Server	*PostgreSQL*	*Oracle*	*MySQL*
00:00:19	*00:02:28*	*00:02:40*	*00:07:31*

- *Piani di esecuzione*
  MS SQL Server:

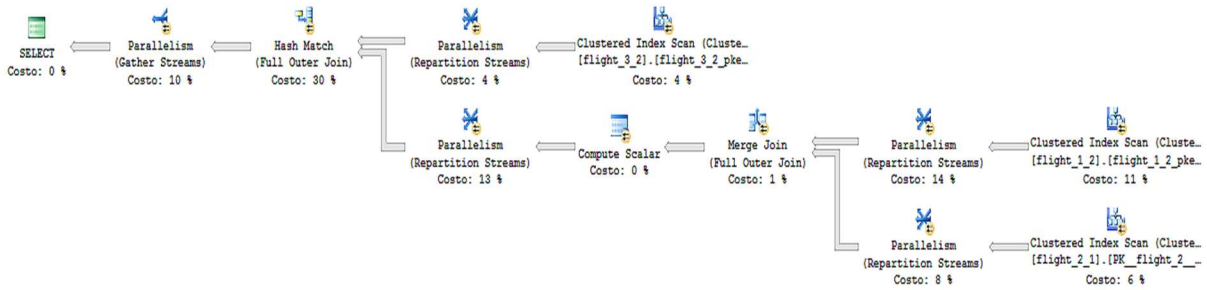


Figura 57

7.1.3.2 Query 2, singola condizione imposta, poco selettiva

```
SELECT *
FROM flight_1_2 f12
FULL OUTER JOIN flight_2_1 f21 ON f21.id=f12.id
FULL OUTER JOIN flight_3_2 f32 ON f32.id=COALESCE(f12.id, f21.id)
WHERE f12.uniquecarrier<>'AA'
```

Versione per MySQL:

```
SELECT f21. * , f32. * , f12. *
FROM flight_1_2 f12
LEFT OUTER JOIN flight_2_1 f21 ON f21.id=f12.id
LEFT OUTER JOIN flight_3_2 f32 ON f32.id=f12.id
WHERE f12.uniquecarrier<>'AA'
UNION
SELECT f21. * , f32. * , f12. *
FROM flight_2_1 f21
LEFT OUTER JOIN flight_1_2 f12 ON f21.id=f12.id
LEFT OUTER JOIN flight_3_2 f32 ON f32.id=f21.id
WHERE f12.uniquecarrier<>'AA'
UNION
SELECT f21. * , f32. * , f12. *
FROM flight_3_2 f32
LEFT OUTER JOIN flight_2_1 f21 ON f32.id=f21.id
LEFT OUTER JOIN flight_1_2 f12 ON f12.id=f32.id
WHERE f12.uniquecarrier<>'AA'
```

- *Righe ritornate: **812,639***
- *Tempi di esecuzione:*

MS SQL Server	*PostgreSQL*	*Oracle*	*MySQL*
00:00:10	*00:01:19*	*00:01:30*	*00:04:55*

7.1.3.3 Query 3, singola condizione imposta, restrittiva

```
SELECT *
FROM flight_1_2 f12
FULL OUTER JOIN flight_2_1 f21 ON f21.id=f12.id
FULL OUTER JOIN flight_3_2 f32 ON f32.id=COALESCE(f12.id, f21.id)
WHERE f12.uniquecarrier='AA'
```

Versione per MySQL:

```
SELECT f21. * , f32. * , f12. *
FROM flight_1_2 f12
LEFT OUTER JOIN flight_2_1 f21 ON f21.id=f12.id
LEFT OUTER JOIN flight_3_2 f32 ON f32.id=f12.id
WHERE f12.uniquecarrier='AA'
UNION
SELECT f21. * , f32. * , f12. *
FROM flight_2_1 f21
LEFT OUTER JOIN flight_1_2 f12 ON f21.id=f12.id
LEFT OUTER JOIN flight_3_2 f32 ON f32.id=f21.id
WHERE f12.uniquecarrier='AA'
UNION
SELECT f21. * , f32. * , f12. *
FROM flight_3_2 f32
LEFT OUTER JOIN flight_2_1 f21 ON f32.id=f21.id
LEFT OUTER JOIN flight_1_2 f12 ON f12.id=f32.id
WHERE f12.uniquecarrier='AA'
```

- *Righe ritornate:* ***16,395***
- *Tempi di esecuzione:*

MS SQL Server	*PostgreSQL*	*Oracle*	*MySQL*
00:00:00.5	*00:00:04.6*	*00:00:20*	*00:00:05*

- *Piani di esecuzione*

  MS SQL Server

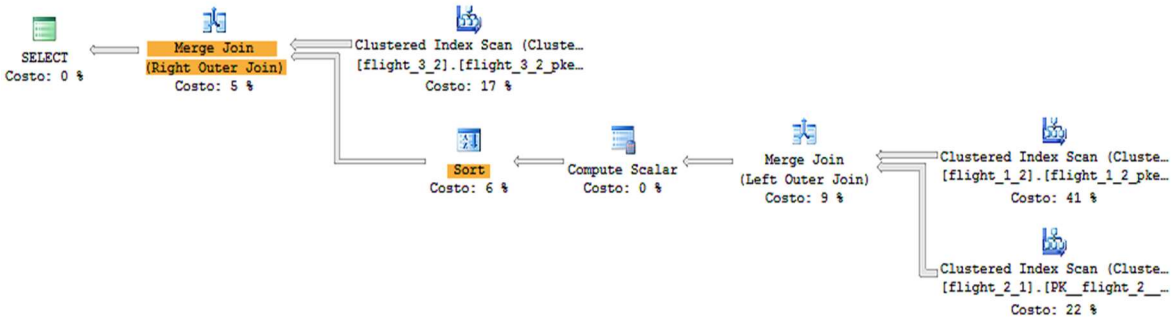


Figura 58

7.1.3.4 Query 4, due condizioni imposte, di cui una selettiva

```
SELECT *
FROM flight_1_2 f12
FULL OUTER JOIN flight_2_1 f21 ON f21.id=f12.id
FULL OUTER JOIN flight_3_2 f32 ON f32.id=COALESCE(f12.id, f21.id)
WHERE f12.uniquecarrier<>'AA'
AND f32.nasdelay='0'
```

Versione per MySQL:

```
SELECT f21. * , f32. * , f12. *
FROM flight_1_2 f12
LEFT OUTER JOIN flight_2_1 f21 ON f21.id=f12.id
LEFT OUTER JOIN flight_3_2 f32 ON f32.id=f12.id
WHERE f12.uniquecarrier<>'AA'
AND f32.nasdelay='0'
UNION
SELECT f21. * , f32. * , f12. *
FROM flight_3_2 f32
LEFT OUTER JOIN flight_2_1 f21 ON f32.id=f21.id
LEFT OUTER JOIN flight_1_2 f12 ON f12.id=f32.id
WHERE f12.uniquecarrier<>'AA'
AND f32.nasdelay='0'
```

- *Righe ritornate:* ***19,145***
- *Tempi di esecuzione:*

MS SQL Server	*PostgreSQL*	*Oracle*	*MySQL*
00:00:00.44	*00:00:03*	*00:00:47*	*00:00:06*

7.1.3.5 Query 5, tre condizioni imposte, poco selettive

```
SELECT *
FROM flight_1_2 f12
FULL OUTER JOIN flight_2_1 f21 ON f21.id=f12.id
FULL OUTER JOIN flight_3_2 f32 ON f32.id=COALESCE(f12.id, f21.id)
WHERE f12.uniquecarrier<>'AA'
AND f32.nasdelay='0'
AND f21.deptime > '30'
```

Versione per MySQL:

```
SELECT f21. * , f32. * , f12. *
FROM flight_3_2 f32
LEFT OUTER JOIN flight_2_1 f21 ON f32.id=f21.id
LEFT OUTER JOIN flight_1_2 f12 ON f12.id=f32.id
WHERE f12.uniquecarrier<>'AA'
AND f21.deptime > '30'
AND f32.nasdelay='0'
```

- *Righe ritornate:* ***134***
- *Tempi di esecuzione:*

MS SQL Server	*PostgreSQL*	*Oracle*	*MySQL*
00:00:00.54	*00:00:00.84*	*00:00:21.45*	*00:00:03.8*

7.1.3.6 Comparazione

Innanzitutto per SQL Server è stato ulteriormente confermato il fatto che in generale se sono specificate condizioni poco selettive si tende a preferire il piano parallelo, mentre se sono condizioni selettive, il fatto che permettano alle operazioni successive di lavorare con un numero basso di righe, già di per sé diminuisce i costi a tal punto da rendere sconveniente il piano seriale, che per raggiungere dei benefici deve però introdurre dei costi di gestione.
Viene confermato anche il fatto che in merito alle Figura 6 e Figura 37 è stato scelto il piano seriale perché il parallelo comporta dei costi aggiuntivi per la gestione che superano i benefici, e non perché siano presenti dei problemi con statistiche non aggiornate.
Dalla Figura 57 infatti si può notare che se i costi associati agli Exchange Operators si mantengono bassi, viene prediletto il piano parallelo e tali costi risultano inferiori quando sono coinvolte tabelle di dimensioni non eccessive.
In poche parole, con tabelle più piccole si hanno costi inferiori per la gestione del parallelismo e quindi prevalgono i benefici introdotti dal parallelismo, quali le diminuzioni dei costi dei join e delle altre operazioni.
Si riportano i costi associati agli Exchange Operators che compaiono in Figura 57, comparati con quelli relativi alle query del primo e del secondo gruppo in cui era stato eseguito lo scambio tra le relazioni più grandi e quelle più piccole:

COSTI AGGIUNTIVI PER L'ESECUZIONE PARALLELA (EXCHANGE OPERATORS):

Figura 57	Gruppo 1	Gruppo 2
REPARTITION STREAMS SU FLIGHT_3_2: 2,13052.	**REPARTITION STREAMS SU FLIGHT_2_3**: 23,9375.	**REPARTITION STREAMS SU FLIGHT_1_1**: 9,9352.
REPARTITION STREAMS SU FLIGHT_1_2: 8,26701.	**REPARTITION STREAMS SU FLIGHT_1_2:** 8,26701.	**REPARTITION STREAMS SU FLIGHT_2_1**: 4,40895.
REPARTITION STREAMS SU FLIGHT_2_1: 4,40895.	**REPARTITION STREAMS SU FLIGHT_1_1:** 19,9674.	**REPARTITION STREAMS SU FLIGHT_1_2**: 8,26701.
REPARTITION STREAMS SECONDO SOTTOALBERO: 7,4478.	**REPARTITION STREAMS PRIMO SOTTOALBERO**: 19,908.	**REPARTITION STREAMS PRIMO SOTTOALBERO**: 7,4478.
GATHER STREAMS: 5,6323.	**GATHER STREAMS**: 78,995.	**GATHER STREAMS**: 25,46.
TOT: 27.88658	TOT: 151.07491	TOT: 55.51896

Tabella 3

Dai dati riportati si può notare che tra tutte le operazioni citate quella il cui costo diminuisce maggiormente con la diminuzione delle dimensioni delle tabelle coinvolte è Gather Streams, che si occupa di elaborare i vari flussi di input per produrre un unico flusso di output di record derivati dalla combinazione dei flussi di input.
Per la prima volta inoltre si può notare un ampio divario in termini di costi associati tra il costo della query di Figura 57 e la stessa eseguita in serie; l'esecuzione parallela infatti produce una riduzione del costo complessivo di 22.8353, pari al 28.45%. Caso analogo per la Query 7.1.3.2, in relazione alla quale si registra un calo del 31.6% del costo.
Questo dimostra che in effetti i costi delle operazioni di gestione del parallelismo aumentano all'aumentare delle dimensioni delle tabelle, fino ad arrivare a casi in cui sono talmente elevati che, in relazione a query in cui il parallelismo apporta pochi benefici perché o non sono state inserite condizioni o se lo sono, si tratta di condizioni poco selettive, è comunque più conveniente il piano seriale.
Se però si considerano delle relazioni di dimensioni minori i costi degli Exchange Operators risultano inferiori, rendendo allora effettivamente utilizzabile e pienamente conveniente l'esecuzione parallela.
Il che non significa che quando si hanno tabelle di dimensioni non elevate e il costo associato alla query superi il Cost Threshold for Parallelism automaticamente venga scelto il piano parallelo; infatti, anche se i costi per la gestione dell'esecuzione parallela sono inferiori quando si hanno relazioni piccole, comunque sono da considerare e possono avere anche un forte impatto sul costo complessivo della query, e se il numero di righe da suddividere tra i diversi threads è molto ridotto in realtà il vantaggio di sfruttare più threads per completare le operazioni c'è ma è piuttosto basso e non quindi tale da superare i costi di gestione; non vi è giovamento a suddividere poche righe tra diversi threads, in quanto possono essere processate da un unico thread - con prestazioni leggermente peggiori - evitando i, seppur più bassi, costi di gestione.
Infatti si ha che il valore predefinito per l'opzione MAXDOP è pari a 8, il che significa che le operazioni eseguite in parallelo prevedono il coinvolgimento di 8 threads per essere portate a termine, che si suddividono non in maniera uguale le righe interessate. Come detto in precedenza, in relazione al gruppo1, il numero di threads (8) è da considerare per singola

operazione e non associato al piano di esecuzione globale.
Con maggior precisione in realtà i processi sono 9, in quanto è sempre presente un ulteriore thread (Thread 0) che è impiegato per la sincronizzazione degli altri threads e raccoglie i dati di output dei *worker threads*; nel suo caso il numero associato alla voce Actual Rows è pari a zero perché non esegue data fetching. [9]
Una situazione tipica di suddivisione delle righe in un'operazione eseguita in parallelo è descritta nella figura sotto riportata (relativa a Query 7.1.3.2, operazione Repartition Streams sul secondo sottoalbero):

Numero effettivo di righe	812639
Thread 0	0
Thread 1	101720
Thread 2	101143
Thread 3	101512
Thread 4	101849
Thread 5	101589
Thread 6	101665
Thread 7	101407
Thread 8	101754

Da tali considerazioni quindi si può concludere che se le righe sono esigue sarebbe sconveniente suddividerle tra ben 8 threads, quando uno unico può processarle individualmente risparmiando i costi di gestione.
Inoltre altro elemento che fa sì che possa non essere scelto automaticamente il piano parallelo quando gli Exchange Operators mantengono un costo basso, è che ci sono alcuni operatori che con l'utilizzo del parallelismo vedono il proprio costo associato aumentare, invece di diminuire, e questo, in una situazione in cui il parallelismo non riduce drasticamente i costi complessivi, può far propendere per il piano seriale.
Sicuramente dai piani di esecuzione riportati si può affermare che una di queste operazione è Sort, utilizzata solitamente in concomitanza con il Merge Join. Esaminando infatti i piani di esecuzione si può notare che solitamente quando è presente il Merge Join, che implica l'operazione di Sort, non viene eseguito il piano parallelo; questo perché se si analizzano i costi relativi si ha, per citarne alcuni:

- Costo Sort *in serie* su righe 16,395: 0.982 (6%).
- Costo Sort *in parallelo* su righe 29,598: 3.7512 (14%).
- Costo Sort *in serie* su righe 108,379: 8.0015 (20%).
- Costo Sort *in parallelo* su righe 67,614: 9.8556 (13%).

Per quanto riguarda i filtri anche in questi esempi SQL Server e PostgreSQL agiscono in maniera simile; imponendo infatti due condizioni selettive, si può notare come entrambi operino diversamente da quanto già visto nel gruppo 2, nel quale era stata applicata subito la condizione maggiormente restrittiva mentre la seconda veniva applicata in seguito al join tra le due tabelle (per SQL Server mediante l'operatore Filter).
In questo caso invece sono applicate direttamente entrambe le condizioni e successivamente le due tabelle sono unite mediante Inner Join; scompare quindi l'operatore Filter. Questa scelta è

stata determinata dal fatto che la seconda condizione restrittiva è in questo caso applicata su una tabella di dimensioni inferiori (499,799 righe totali), per cui l'applicazione diretta del filtro risulta meno onerosa perché devono essere analizzate molte meno righe e allo stesso tempo apporta dei vantaggi, quali il fatto che viene eseguito un Inner Join, già di per sé operazione meno costosa, ed esso è realizzato considerando un numero decisamente inferiore di righe, altro aspetto positivo. Quindi se le condizioni sono poco restrittive entrambi le applicano tutte direttamente; se una è restrittiva e l'altra no viene applicata solo quella più restrittiva immediatamente mentre la seconda è applicata in un secondo momento mediante un filtro apposito; se entrambe le condizioni sono restrittive e le tabelle hanno elevate dimensioni (829,034 righe) è comunque applicata immediatamente solo quella che riduce maggiormente il numero di righe e infine se le condizioni sono selettive e applicate su tabelle di dimensioni medie (499,799 righe) vengono applicate direttamente entrambe, in modo da realizzare un Inner Join.
Anche per Oracle XE si presenta una situazione simile. Da quanto visto nelle considerazioni dei gruppi precedenti, Oracle XE in presenza di condizioni poco selettive - che siano due o tre - decide di applicarle tutte immediatamente tramite Table Access, adottando una strategia simile a SQL Server e PostgreSQL; quando invece tra le condizioni imposte ve n'è almeno una restrittiva, le applica tutte come penultima operazione prima della SELECT, e in questo differisce da quanto invece eseguito da SQL Server e PostgreSQL; osservando invece i risultati ottenuti nel gruppo 3, si nota che Oracle XE torna ad agire analogamente a SQL Server e PostgreSQL: quando le condizioni sono applicate su tabelle di piccole/medie dimensioni (circa 380 mila/500mila righe), indipendentemente dalla natura selettiva, le applica subito contestualmente a Table Scan, dimostrando nuovamente che risulta conveniente l'applicazione prima della SELECT solo su tabelle di grandi dimensioni.

7.1.3.7 Comparazione con MySQL

Rimangono valide tutte le considerazioni precedenti; non si riscontrano differenze rispetto quanto già analizzato negli altri gruppi. Per completezza nei confronti, si riportano nelle considerazioni finali soltanto i tempi di esecuzione delle query riformulate eseguite negli altri DBMS, confrontati con quelli di MySQL; per quanto riguarda i piani di esecuzione non sono riportati in quanto analoghi a quelli già in precedenza esaminati.

7.1.4 Gruppo 4

Si riportano esclusivamente le query eseguite e i relativi tempi di esecuzione, in quanto per tutti i DBMS considerati i piani di esecuzione si mantengono identici nella struttura a quelli precedentemente illustrati nel gruppo 3 (MS SQL Server) e nei gruppi 1 e 2 (PostgreSQL, Oracle XE e MySQL). Non vi sono significative osservazioni da aggiungere.

7.1.4.1 Query 1, nessuna condizione imposta

```
SELECT *
FROM flight_2_1 f21
FULL OUTER JOIN flight_1_3 f13 ON f13.id=f21.id
FULL OUTER JOIN flight_3_1 f31 ON f31.id=COALESCE(f13.id, f21.id)
```

Versione per MySQL:

```
SELECT f21. * , f13. * , f31. *
FROM flight_2_1 f21
LEFT OUTER JOIN flight_1_3 f13 ON f13.id=f21.id
LEFT OUTER JOIN flight_3_1 f31 ON f31.id=f21.id
UNION
SELECT f21. * , f13. * , f31. *
FROM flight_1_3 f13
LEFT OUTER JOIN flight_2_1 f21 ON f21.id=f13.id
LEFT OUTER JOIN flight_3_1 f31 ON f31.id=f13.id
UNION
SELECT f21. * , f13. * , f31. *
FROM flight_3_1 f31
LEFT OUTER JOIN flight_2_1 f21 ON f31.id=f21.id
LEFT OUTER JOIN flight_1_3 f13 ON f13.id=f31.id
```

- *Righe ritornate:* ***803,290***
- *Tempi di esecuzione:*

MS SQL Server	*PostgreSQL*	*Oracle*	*MySQL*
00:00:09	*00:01:13*	*00:02:40*	*00:01:52*

7.1.4.2 Query 2, singola condizione imposta, selettiva

```
SELECT *
FROM flight_2_1 f21
FULL OUTER JOIN flight_1_3 f13 ON f13.id=f21.id
FULL OUTER JOIN flight_3_1 f31 ON f31.id=COALESCE(f13.id, f21.id)
WHERE f21.deptime > '30'
```

Versione per MySQL:

```
SELECT f21. * , f13. * , f31. *
FROM flight_2_1 f21
LEFT OUTER JOIN flight_1_3 f13 ON f13.id=f21.id
LEFT OUTER JOIN flight_3_1 f31 ON f31.id=f21.id
WHERE f21.deptime > '30'
UNION
SELECT f21. * , f13. * , f31. *
FROM flight_1_3 f13
LEFT OUTER JOIN flight_2_1 f21 ON f21.id=f13.id
LEFT OUTER JOIN flight_3_1 f31 ON f31.id=f13.id
WHERE f21.deptime > '30'
UNION
SELECT f21. * , f13. * , f31. *
```

```
FROM flight_3_1 f31
LEFT OUTER JOIN flight_2_1 f21 ON f31.id=f21.id
LEFT OUTER JOIN flight_1_3 f13 ON f13.id=f31.id
WHERE f21.deptime > '30'
```

- *Righe ritornate:* ***188,491***
- *Tempi di esecuzione:*

MS SQL Server	*PostgreSQL*	*Oracle*	*MySQL*
00:00:02.3	*00:00:17*	*00:00:23*	*00:00:15*

7.1.4.3 Query 3, due condizioni imposte, selettive

```
SELECT *
FROM flight_2_1 f21
FULL OUTER JOIN flight_1_3 f13 ON f13.id=f21.id
FULL OUTER JOIN flight_3_1 f31 ON f31.id=COALESCE(f13.id, f21.id)
WHERE f21.deptime > '30'
AND f13.flightnum < '300'
```

Versione per MySQL:

```
SELECT f21. * , f13. * , f31. *
FROM flight_2_1 f21
LEFT OUTER JOIN flight_1_3 f13 ON f13.id=f21.id
LEFT OUTER JOIN flight_3_1 f31 ON f31.id=f21.id
WHERE f21.deptime > '30'
AND f13.flightnum < '300'
UNION
SELECT f21. * , f13. * , f31. *
FROM flight_3_1 f31
LEFT OUTER JOIN flight_2_1 f21 ON f31.id=f21.id
LEFT OUTER JOIN flight_1_3 f13 ON f13.id=f31.id
WHERE f21.deptime > '30'
AND f13.flightnum < '300'
```

- *Righe ritornate:* ***2,163***
- *Tempi di esecuzione:*

MS SQL Server	*PostgreSQL*	*Oracle*	*MySQL*
00:00:00.02	*00:00:00.7*	*00:00:07*	*00:00:03.5*

7.1.4.4 Query 4, tre condizioni imposte, selettive

```
SELECT *
FROM flight_2_1 f21
FULL OUTER JOIN flight_1_3 f13 ON f13.id=f21.id
FULL OUTER JOIN flight_3_1 f31 ON f31.id=COALESCE(f13.id, f21.id)
WHERE f21.deptime > '30'
AND f13.flightnum < '300'
AND f31.cancellationcode='A'
```

Versione per MySQL:

```
SELECT f21. * , f13. * , f31. *
FROM flight_3_1 f31
LEFT OUTER JOIN flight_2_1 f21 ON f31.id=f21.id
```

```
LEFT OUTER JOIN flight_1_3 f13 ON f13.id=f31.id
WHERE f21.deptime > '30'
AND f13.flightnum < '300'
AND f31.cancellationcode='A'
```

- *Righe ritornate:* ***26***
- *Tempi di esecuzione:*

MS SQL Server	*PostgreSQL*	*Oracle*	*MySQL*
00:00:00.143	*00:00:00.57*	*00:00:06*	*00:00:02*

7.2 INTERROGAZIONI CON APPOSITI INDICI

Gli indici sono speciali *lookup tables* che il *search engine* del database può impiegare per velocizzare il recupero dei dati. Per tale motivo possono rendere più efficienti le query basate sull'operazione di SELECT, e le clausole WHERE, mentre comportano un rallentamento nelle istruzioni di INSERT e UPDATE. [10]

Per un confronto più completo dei DBMS considerati, si riportano alcune delle query precedentemente immesse, svolte sfruttando indici appositamente creati.

Anche in questo caso si mantiene una distinzione basata sulle dimensioni delle tabelle coinvolte, in modo da registrare eventuali differenze nelle strategie adottate a seconda della cardinalità considerata.

Si riportano i dati e alcuni piani di esecuzione ottenuti - quelli che apportano nuove e significative informazioni - operando contestualmente un confronto con le medesime query svolte precedentemente senza indici.

7.2.1 Singolo indice su *flight_2_3.deptime* (6,351,272 righe):

```
CREATE INDEX Idx1 ON flight_2_3 (deptime);
```

Query poco selettiva:

```
SELECT *
FROM flight_2_3 f23
FULL OUTER JOIN flight_1_1 f11 ON f11.id=f23.id
FULL OUTER JOIN flight_1_2 f12 ON f12.id=COALESCE(f11.id, f23.id)
WHERE f23.deptime>'1500'
```

- Rows affected: **4,184,487**
- Rows removed by filter: **2,166,786 (34.11%)**

	SQL Server	*PostgreSQL*	*Oracle*	*MySQL*
Tempo di esecuzione con indice	*00:00:49*	*00:07:27*	*00:45:02*	*00:38:21*
Tempo di esecuzione senza indice	*00:00:49*	*00:07:29*	*00:45:42*	*00:38:26*
Miglioramento	-	2 secondi (0.4%)	40 secondi (1.4%)	5 secondi (0.3%)

Tabella 4

Per SQL Server non si registra alcun miglioramento in termini di prestazioni; la presenza dell'indice sul campo *deptime* di flight_2_3 non comporta variazioni nel piano di esecuzione, che si mantiene esattamente uguale a quello presentato in Figura 11.
Considerando i dati di PostgreSQL, Oracle e MySQL si registra un miglioramento di 2 secondi per il primo, 40 secondi per il secondo e 5 secondi per il terzo, ma si tratta di risultati dovuti a fattori diversi dalla creazione dell'indice, in quanto anche in questo caso il piano di esecuzione prodotto da ognuno dei tre DBMS non presenta minime differenze rispetto quello in Figura 12 (PostgreSQL), Figura 13 (Oracle XE) o Figura 14 (MySQL).

Query selettiva:

```
SELECT *
FROM flight_2_3 f23
FULL OUTER JOIN flight_1_1 f11 ON f11.id=f23.id
FULL OUTER JOIN flight_1_2 f12 ON f12.id=COALESCE(f11.id, f23.id)
WHERE f23.deptime<'1500'
```

- Rows affected: **2,156,206**
- Rows removed by filter: **4,195,066 (66.05%)**

	SQL Server	*PostgreSQL*	*Oracle*	*MySQL*
Tempo di esecuzione con indice	*00:00:27*	*00:03:49*	*00:38:58*	*00:21:59*
Tempo di esecuzione senza indice	*00:00:27*	*00:03:45*	*00:39:41*	*00:22:04*
Miglioramento	-	(Peggioramento) 4 secondi (1.7%)	43 secondi (1.8%)	5 secondi (0.34%)

Tabella 5

Nonostante la condizione ora imposta sia selettiva, determinando l'eliminazione del 66.05% di righe, non lo è a sufficienza per far propendere i query optimizer di ciascun DBMS ad optare per uno scan method differente rispetto quelli già adottati in precedenza. I piani di esecuzione infatti non presentano ancora una volta elementi differenti rispetto i precedenti. Si riscontrano leggere variazioni nei tempi di esecuzione registrati, ma la loro presenza non dipende dall'indice su *flight_2_3.deptime*.

Query altamente selettiva:

```
SELECT *
FROM flight_2_3 f23
FULL OUTER JOIN flight_1_1 f11 ON f11.id=f23.id
FULL OUTER JOIN flight_1_2 f12 ON f12.id=COALESCE(f11.id, f23.id)
WHERE f23.deptime<'100'
```

- Rows affected: **879**
- Rows removed by filter: **6,350,393 (99%)**

	SQL Server	*PostgreSQL*	*Oracle*	*MySQL*
Tempo di esecuzione con indice	*00:00:00.310*	*00:00:01.1*	*00:00:48.35*	*00:00:06.87/ 00:00:03.61 (index-only)*
Tempo di esecuzione senza indice	*00:00:05.502*	*00:00:03.6*	*00:01:04*	*00:00:46.922*
Miglioramento	5.19 secondi (94.37%)	2.5 secondi (69.44%)	15.65 secondi (24.4%)	40.05 secondi (85.3%)/ 43.3 secondi (92.3%)

Tabella 6

L'indice definito unicamente sulla colonna *deptime* induce solo MySQL e PostgreSQL a modificare la strategia utilizzata, apportando così diminuzioni nei tempi di esecuzione; i piani di esecuzione, riportati sotto, risultano pertanto differenti rispetto gli usuali:

```
1  QUERY PLAN
2  Hash Left Join  (cost=104329.99..205634.06 rows=52364 width=161) (actual time=12334.337..26641.639 rows=879 loops=1)
3    Hash Cond: (COALESCE(f11.id, f23.id) = f12.id)
4    ->  Hash Right Join  (cost=67390.73..157430.79 rows=52364 width=104) (actual time=481.386..18480.209 rows=879 loops=1)
5          Hash Cond: (f11.id = f23.id)
6          ->  Seq Scan on flight_1_1 f11  (cost=0.00..42699.47 rows=2006447 width=56) (actual time=0.057..8919.567 rows=2006447 loops=1)
7          ->  Hash  (cost=66275.18..66275.18 rows=52364 width=48) (actual time=10.279..10.279 rows=879 loops=1)
8                Buckets: 65536  Batches: 2  Memory Usage: 548kB
9                ->  Bitmap Heap Scan on flight_2_3 f23  (cost=982.25..66275.18 rows=52364 width=48) (actual time=0.439..5.709 rows=879 loops=1)
10                     Recheck Cond: ((deptime)::text < '100'::text)
11                     Heap Blocks: exact=863
12                     ->  Bitmap Index Scan on idx1  (cost=0.00..969.16 rows=52364 width=0) (actual time=0.320..0.320 rows=879 loops=1)
13                           Index Cond: ((deptime)::text < '100'::text)
14   ->  Hash  (cost=17670.34..17670.34 rows=829034 width=57) (actual time=7984.320..7984.320 rows=829034 loops=1)
15         Buckets: 65536  Batches: 32  Memory Usage: 2830kB
16         ->  Seq Scan on flight_1_2 f12  (cost=0.00..17670.34 rows=829034 width=57) (actual time=0.034..3976.324 rows=829034 loops=1)
```

Figura 59

Il piano di esecuzione prodotto da PostgreSQL, in relazione all'analisi di flight_2_3, presenta due differenti step: il primo ad essere eseguito visita l'indice Idx1, costruito sul campo deptime di flight_2_3, con lo scopo di individuare le locazioni delle righe per le quali esiste un match con la condizione espressa; il secondo step preleva le righe individuate dalla tabella stessa [11].

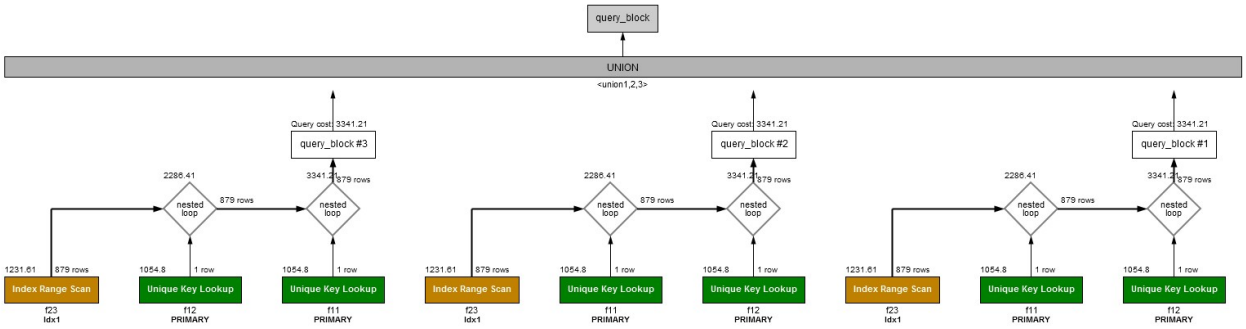


Figura 6o

MySQL mantiene la struttura consueta, già presentata e analizzata, con l'importante differenza che tutte le operazioni di Full Table Scan sono sostituite da Index Range Scan: si tratta di scan parziali sull'indice definito su flight_2_3 e il codice colori supportato da MySQL dimostra immediatamente che si tratta di operatori meno costosi rispetto i precedenti, seppure non siano tra i più economici, ma si attestano su un livello medio.
Tramite le operazioni di Index Range Scan, MySQL è in grado di trovare rapidamente le righe interessate, senza scorrere completamente l'intero dataset, ma basando la ricerca proprio sul range definito dalla clausola WHERE; in questo caso il range è definito completamente dalla stessa clausola, ma nel caso in cui fossero presenti dei predicati che portassero a sovrapposizione di range diversi, MySQL è in grado di ricondurli a intervalli separati in quanto supporta la possibilità di ottimizzare diversi range specificati [12].

SQL Server e Oracle XE non utilizzano l'indice così definito, continuando a generare dei piani di esecuzione analoghi ai precedenti; affinché venga effettivamente sfruttato per il reperimento delle informazioni in un tempo inferiore, devono essere inserite nella definizione dell'indice stesso anche le altre colonne richieste dalla SELECT; in questo modo sono in grado, tramite l'indice, non solo di valutare la condizione di WHERE, ma anche di evitare completamente accessi alle tabella flight_2_3, perché il Query Optimizer può individuare tutti i valori delle colonne richieste dalla SELECT nell'indice stesso. In questo modo, la quantità di operazioni di I/O su disco è inferiore dato che non viene eseguito alcun accesso ai dati delle tabelle e quindi si hanno migliori prestazioni. Indici così definiti hanno però lo svantaggio di utilizzare maggiore memoria e incrementare lo sforzo necessario per la manutenzione in caso di operazioni di update. Si parla in questi casi di *Index Only Scan* [13].

Sintassi per SQL Server:
```
CREATE NONCLUSTERED INDEX Idx2 ON flight_2_3 (deptime)
INCLUDE (id, crsdeptime, arrtime, crsarrtime);
```
Sintassi per Oracle XE:
```
CREATE INDEX Idx2 ON flight_2_3 (deptime,id, crsdeptime, arrtime, crsarrtime);
```

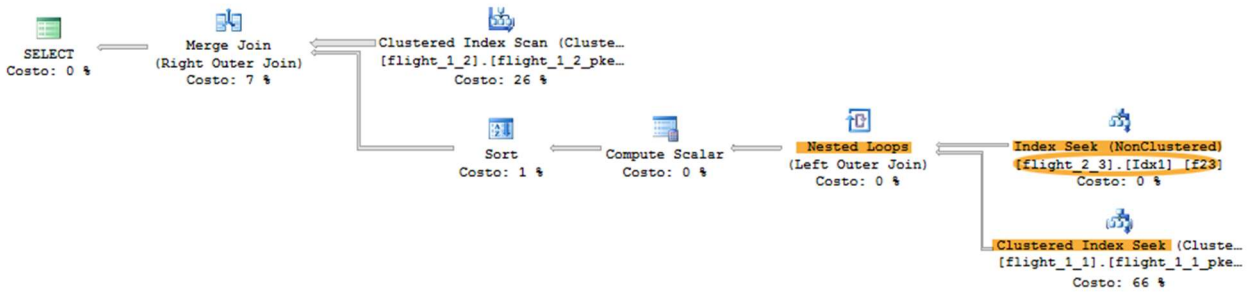


Figura 61

SQL Server impiega l'indice creato (Idx1) per realizzare un Nested Loop Join indicizzato tra flight_2_3 e flight_1_1. In questo modo si prelevano le sole righe interessate, che soddisfano la condizione, senza dover scorrere l'intero indice, ottenendo un costo approssimabile a zero (0.0446391); il costo, direttamente proporzionale alla quantità di righe ritornate, si mantiene particolarmente basso in quanto il numero di righe che soddisfano la condizione imposta è a sua volta ridotto.

Operation	Object	Optimizer	Cost	Cardinality	Bytes
◢ SELECT STATEMENT		ALL_ROWS	17,605	40,867	15,652,061
◢ HASH JOIN (OUTER)			17,605	40,867	15,652,061
◢ VIEW	VW_FOJ_0		12,165	40,867	13,240,908
◢ HASH JOIN (OUTER)			12,165	40,867	4,577,104
INDEX (RANGE SCAN)	IDX2	ANALYZED	361	40,867	2,125,084
TABLE ACCESS (FULL)	FLIGHT_1_1	ANALYZED	4,839	2,006,446	120,386,760
TABLE ACCESS (FULL)	FLIGHT_1_2	ANALYZED	2,002	829,034	48,913,006

Figura 62

Oracle XE accede alle informazioni mediante operazione di Index Range Scan sull'indice Idx2 appena creato. In questo modo preleva un set di righe (le uniche soddisfacenti la condizione) direttamente dall'indice stesso.

Applicando poi gli indici definiti su più colonne anche su PostgreSQL e MySQL, per il primo non cambiano né il piano né il tempo di esecuzione, mentre per MySQL viene mantenuta la struttura del piano di esecuzione, con l'unica eccezione che le operazioni di Index Range Scan sono svolte sul nuovo indice, quindi, non dovendo recuperare le informazioni dalla tabella originaria, ma potendole prelevare direttamente dall'indice, i costi associati a tali operazioni si riducono, producendo una diminuzione del 92.3% dei tempi di esecuzione.

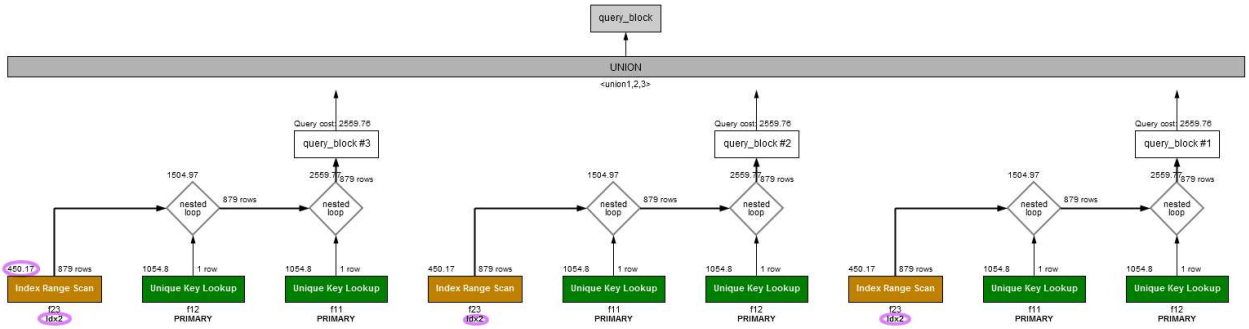


Figura 63

7.2.2 Singolo indice su *flight_1_1.uniquecarrier* (2,006,447 righe)

```
CREATE INDEX Idx3 ON flight_1_1 (uniquecarrier,id, flightnum, tailnum, origin,
dest, distance);
```

Sintassi per SQL Server:

```
CREATE NONCLUSTERED INDEX Idx ON flight_1_1 (uniquecarrier)
INCLUDE (id, flightnum, tailnum, origin, dest, distance);
```

Query selettiva:

```
SELECT *
FROM flight_1_1 f11
FULL OUTER JOIN flight_1_2 f12 ON f11.id=f12.id
FULL OUTER JOIN flight_2_1 f21 ON f21.id=COALESCE(f11.id, f12.id)
WHERE f11.uniquecarrier='EV'
```

- Rows affected: **108,379**
- Rows removed by filter: **1,898,068 (94%)**

	SQL Server	*PostgreSQL*	*Oracle*	*MySQL*
Tempo di esecuzione con indice	*00:00:02.306*	*00:00:16.1*	*00:00:32.3*	*00:04:39*
Tempo di esecuzione senza indice	*00:00:03.246*	*00:00:18.3*	*00:00:34.2*	*00:23:12*
Miglioramento	0.94 secondi (28.9%)	2.2 secondi (12%)	1.9 secondi (5.5%)	18 minuti e 51 s (79.9%)

Tabella 7

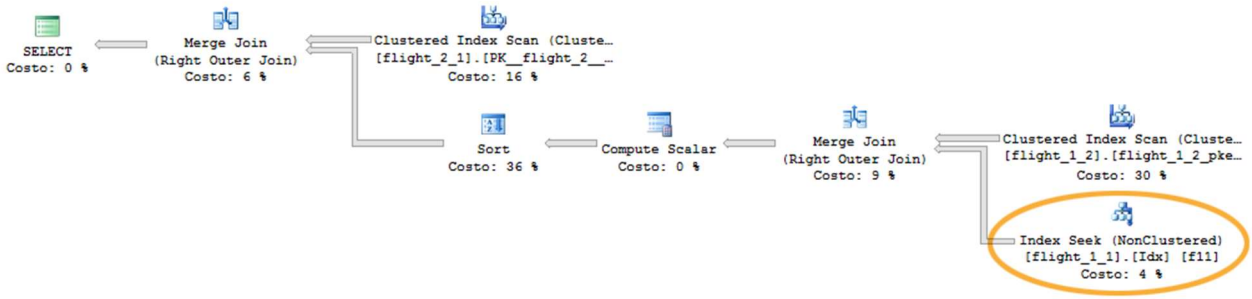


Figura 64

SQL Server utilizza l'indice creato per reperire le informazioni richieste direttamente dall'indice stesso tramite Index Seek; il costo dell'operazione aumenta leggermente rispetto quello precedente (0.860209), in quanto è proporzionale alle righe ritornate dall'operatore, in questo caso in numero superiore (108,379).

7.2.3 Singolo indice su *flight_1_2.uniquecarrier* (829034 righe):

```
CREATE INDEX Idx4 ON flight_1_2 (uniquecarrier,id, flightnum, tailnum, origin,
dest, distance);
```

Sintassi per SQL Server:

```
CREATE NONCLUSTERED INDEX Idx ON flight_1_2 (uniquecarrier)
INCLUDE (id, flightnum, tailnum, origin, dest, distance);
```

Query poco selettiva:

```
SELECT *
FROM flight_1_2 f12
FULL OUTER JOIN flight_2_1 f21 ON f21.id=f12.id
FULL OUTER JOIN flight_3_2 f32 ON f32.id=COALESCE(f12.id, f21.id)
WHERE f12.uniquecarrier<>'AA'
```

- Rows affected: **812639**
- Rows removed by filter: **16395 (2%)**

	SQL Server	*PostgreSQL*	*Oracle*	*MySQL*
Tempo di esecuzione con indice	*00:00:10*	*00:01:19*	*00:01:29*	*00:04:54*
Tempo di esecuzione senza indice	*00:00:10*	*00:01:19*	*00:01:30*	*00:04:55*
Miglioramento	-	-	1 secondo (1.1%)	1 secondo (0.3%)

Tabella 8

Come prevedibile i piani di esecuzione prodotti rimangono invariati rispetto quelli generati prima della definizione del nuovo indice, a causa della natura poco selettiva della condizione, che renderebbe più oneroso uno scan basato sull'indice.

Query altamente selettiva:

```
SELECT *
FROM flight_1_2 f12
FULL OUTER JOIN flight_2_1 f21 ON f21.id=f12.id
FULL OUTER JOIN flight_3_2 f32 ON f32.id=COALESCE(f12.id, f21.id)
WHERE f12.uniquecarrier='AA'
```

- Rows affected: **16,395**
- Rows removed by filter: **812,639 (98%)**

	SQL Server	*PostgreSQL*	*Oracle*	*MySQL*
Tempo di esecuzione con indice	*00:00:01.07*	*00:00:01.8*	*00:00:16.27*	*00:00:02.12*
Tempo di esecuzione senza indice	*00:00:02.11*	*00:00:04.6*	*00:00:20.122*	*00:00:05.343*
Miglioramento	1.04 secondi (49.3%)	2.8 secondi (60.9%)	3.85 secondi (19.1%)	3.22 secondi (60%)

Tabella 9

7.2.4 Due indici: flight_1_2.uniquecarrier (829,034 righe) e flight_2_1.deptime (499,799 righe)

```
CREATE index Idx5 ON flight_2_1 (deptime,id, crsdeptime, arrtime, crsarrtime);
```

Sintassi per SQL Server:

```
CREATE NONCLUSTERED INDEX Idx5 ON flight_2_1 (deptime)
INCLUDE (id, crsdeptime, arrtime, crsarrtime);

SELECT *
FROM flight_1_2 f12
FULL OUTER JOIN flight_2_1 f21 ON f21.id=f12.id
FULL OUTER JOIN flight_3_2 f32 ON f32.id=COALESCE(f12.id, f21.id)
WHERE f12.uniquecarrier='AA'
AND f21.deptime > '9500'
```

- Rows affected: **118**
- Rows removed by filter on flight_1_2: **812639 (98%)**
- Rows removed by filter on flight_2_1: **452251 (90.5%)**

	SQL Server	*PostgreSQL*	*Oracle*	*MySQL*
Tempo di esecuzione con indice	*00:00:00.282*	*00:00:00.098*	*00:00:07.7*	*00:00:00.87*
Tempo di esecuzione senza indice	*00:00:00.328*	*00:00:00.344*	*00:00:08.1*	*00:00:01.4*
Miglioramento	0.046 secondi (86%)	0.246 secondi (71.5%)	0.4 secondi (5%)	0.53 secondi (37.8%)

Tabella 10

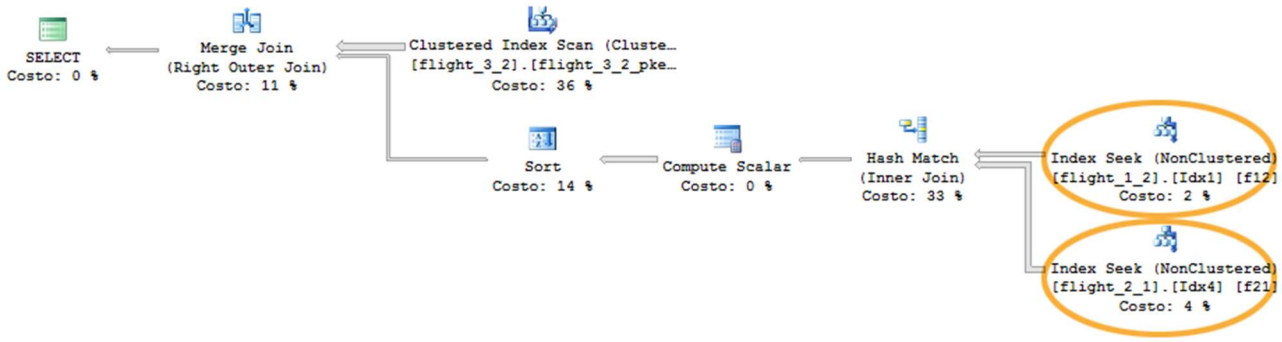


Figura 65

L'unico elemento che varia nel piano di esecuzione prodotto da SQL Server è costituito dall'applicazione dell'operatore Index Seek sugli indici creati in entrambi i rami del secondo sottoalbero. L'effetto ottenuto è quello di ridurre notevolmente il costo complessivo della query, ora pari a 7.65976.

```
1  QUERY PLAN
2  Nested Loop Left Join  (cost=13177.17..21065.91 rows=971 width=150)
3   -> Hash Join  (cost=13176.75..19125.72 rows=971 width=105)
4       Hash Cond: (f21.id = f12.id)
5       -> Bitmap Heap Scan on flight_2_1 f21  (cost=2289.05..8038.33 rows=50662 width=48)
6           Recheck Cond: ((deptime)::text > '9500'::text)
7           -> Bitmap Index Scan on idx5  (cost=0.00..2276.39 rows=50662 width=0)
8               Index Cond: ((deptime)::text > '9500'::text)
9       -> Hash  (cost=10689.07..10689.07 rows=15890 width=57)
10          -> Bitmap Heap Scan on flight_1_2 f12  (cost=795.70..10689.07 rows=15890 width=57)
11              Recheck Cond: ((uniquecarrier)::text = 'AA'::text)
12              -> Bitmap Index Scan on idx4  (cost=0.00..791.72 rows=15890 width=0)
13                  Index Cond: ((uniquecarrier)::text = 'AA'::text)
14  -> Index Scan using flight_3_2_pkey on flight_3_2 f32  (cost=0.42..1.99 rows=1 width=45)
15      Index Cond: (id = COALESCE(f12.id, f21.id))
```

Figura 66

Anche nel piano di esecuzione di PostgreSQL compaiono le operazioni per la ricerca basata su indici in entrambi i rami del primo sottoalbero. Le operazioni eseguite rimangono le stesse già analizzate; l'unico elemento che si discosta dai casi precedenti è la realizzazione del join finale mediante algoritmo di Nested Loop.

Operation	Object	Optimizer	Cost	Cardinality	Bytes
◢ SELECT STATEMENT		ALL_ROWS	5,572	55,466	20,577,886
◢ HASH JOIN (OUTER)			5,572	55,466	20,577,886
◢ VIEW	VW_FOJ_0		2,876	55,466	17,970,984
◢ HASH JOIN			2,876	55,466	6,156,726
INDEX (RANGE SCAN)	IDX4	ANALYZED	544	55,269	3,260,871
TABLE ACCESS (FULL)	FLIGHT_2_1	ANALYZED	1,054	359,570	18,697,640
TABLE ACCESS (FULL)	FLIGHT_3_2	ANALYZED	751	380,530	17,884,910

Figura 67

Oracle XE si comporta diversamente da SQL Server e PostgreSQL: sfrutta infatti unicamente l'indice creato su flight_1_2, ovvero sulla tabella che subisce una maggiore riduzione delle righe, eseguendo la consueta operazione di Index Range Scan, mentre per la tabella flight_2_1 opta per l'operazione Full Access Table, senza considerare l'indice relativamente creato.
Non si tratta di una scelta vincente; analizzando infatti i dati si può notare che per esempio PostgreSQL, utilizzando entrambi gli indici, riesce ad ottenere un incremento percentuale delle prestazioni maggiore rispetto quelli ottenuti nei casi precedenti, mentre per Oracle XE si registra l'incremento inferiore.

I risultati ottenuti evidenziano che, in presenza di un indice consono sulla tabella, creato su campi sui quali sono specificate clausole WHERE, esso è impiegato per realizzare la ricerca basata su indice dei dati richiesti, solamente qualora le query immesse siano altamente selettive, quindi ritornino non più del 10-15% delle righe complessive. Tale scelta è determinata dal fatto che una ricerca basata su indici è solitamente più costosa rispetto l'esecuzione di uno scan completo sulla tabella, e quindi utilizzata quando si ha la certezza di dover prelevare un numero ridotto di righe. Perché vi sia una ricerca basata su indice non è quindi condizione sufficiente la presenza di un indice sulle colonne interessate dalla clausola WHERE; si tratta di un comportamento comunemente scelto da tutti i DBMS considerati.
Un'importante distinzione tra i DBMS considerati è rappresentata dal fatto che sia SQL Server, sia Oracle XE, in presenza di indici definiti unicamente sulle colonne delle clausole WHERE di una query che richiede nella SELECT informazioni contenute in campi differenti da quelli coinvolti nella clausola stessa, elaborano un piano di esecuzione che non prevede di analizzare l'indice creato per il reperimento delle informazioni, ma il query optimizer in entrambi i casi decide di analizzare direttamente la tabella stessa, ignorando la presenza dell'indice. Questo comportamento è adottato anche nei casi di condizioni imposte fortemente restrittive (eliminazione del 98% delle righe), che quindi non richiederebbero un livello intermedio per il recupero dei dati oneroso (consultazione dell'indice ed estrazione delle informazioni da tabella). Sia SQL Server che Oracle XE, quindi, per le query con una tale formulazione, richiedono definizioni di indici che permettano *Index-only Scan*, ovvero permettano di reperire tutti i valori

delle colonne richieste dalla SELECT nell'indice stesso, riducendo le operazioni di I/O su disco e di conseguenza incrementando le prestazioni.
Indici così definiti hanno però lo svantaggio di utilizzare maggiore memoria e incrementare lo sforzo necessario per la manutenzione in caso di operazioni di update.
Al contrario, PostgreSQL e MySQL riescono a sfruttare anche indici definiti solo sulle colonne sulle quali sono imposte condizioni: PostgreSQL mantiene la medesima strategia in entrambe le formulazioni, non presentando alcuna differenza né nei tempi né nei piani di esecuzione; MySQL trae un maggior vantaggio grazie alla creazione di indici per la realizzazione di *Index-only Scan*, fattore che può portare ad una riduzione del 92% dei tempi di esecuzione rispetto un indice definito unicamente sulla colonna coinvolta nella clausola WHERE.

Non si registrano differenze rilevanti in relazione a cambiamenti di cardinalità delle relazioni coinvolte; l'unico aspetto che può variare è costituito dall'impiego di algoritmi di join differenti, scelti in maniera adeguata rispetto le diverse dimensioni delle tabelle considerate.
Se la query è basata su due tabelle sulle quali sono definiti indici per l'esecuzione di *Index-only Scan*, per SQL Server o PostgreSQL la ricerca delle righe interessate avviene per entrambe le tabelle tramite consultazione dell'indice, secondo le specifiche usuali modalità previste dai diversi DBMS; mentre Oracle XE sfrutta solamente l'indice applicato sulla tabella che subisce una maggiore riduzione delle righe dovuta alla maggiore selettività della condizione imposta. Applicando tale strategia Oracle XE registra un incremento percentuale delle performance inferiore.

8 CONCLUSIONI

Si riprendono delle affermazioni esposte nel Capitolo 2, nel quale si erano citati, tra le principali scelte operate dai Query Optimizer di ciascun DBMS, i seguenti punti:

1. quali Scan methods impiegare: Seek o Scan su indici o tabelle;
2. quali algoritmi di join eseguire: Sort Merge, Hash Join o Nested Loop Join;
3. ordine dei join.

In relazione al secondo punto, dai piani di esecuzione prodotti, si sono tratte le seguenti informazioni:

	Hash Join	*Sort Merge Join*	*Nested Loop Join*
MS SQL Server	1. Tabelle non ordinate; 2. Una tabella ordinata, l'altra, da ordinare, di dimensione maggiori di 677,426 righe.	1. Entrambe le relazioni ordinate sulla clustered primary key; 2. Una tabella ordinata, l'altra, da ordinare, di dimensione inferiori a 330,320 righe.	-
PostgreSQL	Tabelle non ordinate, di dimensioni maggiori 220,368 righe	Tabelle non ordinate ma prevista una successiva operazione di ordinamento.	Indicizzato: con *outer input* di dimensione inferiore a 108,379 righe.
Oracle XE	Sempre	-	-
MySQL	-	-	Sempre

Tabella 11

SQL Server, quindi, ricorre a varie tipologie di join per realizzare quanto richiesto dall'interrogazione; si può notare che l'algoritmo prediletto è costituito dal Merge Join, che per SQL Server risulta particolarmente efficiente, in quanto i dataset sono ordinati sulla *clustered primary key*. Per realizzare il primo dei due Full Outer Join specificati, infatti, SQL Server ricorre sempre all'algoritmo di Merge Join, mentre il secondo Full Outer Join è realizzato con algoritmi differenti a seconda della dimensione della tabella proveniente dal primo Outer Join, non ordinata. Dai dati a disposizione, si può affermare che quando tale tabella conta meno di 330,320 righe essa viene ordinata per poi eseguire un Merge Join, con l'altra già ordinata *sulla clustered primary key*; nel caso in cui invece la cardinalità sia maggiore (677,426 righe), il query optimizer opta per un Hash Match. In nessuno dei casi analizzati ha fatto ricorso ad un algoritmo di Nested Loop Join.

Focalizzandosi invece su PostgreSQL si può notare come, non disponendo come SQL Server di tecniche di parallelismo e indici cluster, tutti i piani di esecuzione dei quattro gruppi rimangano fondamentalmente simili tra loro. Indipendentemente quindi dal numero di righe che compongono le relazioni coinvolte, la struttura dei query plan rimane la stessa: i full outer join

vengono sempre realizzati mediante Hash join, ad eccezione di quando, imponendo delle condizioni restrittive, non si presentano le condizioni favorevoli per un Nested Loop Join indicizzato, in quanto le righe di una delle due relazioni sono abbastanza esigue da poter essere impiegate come outer input (inferiori a 108,379 righe).
In due soli casi PostgreSQL ha fatto ricorso all'algoritmo di Merge Join, ovvero quando era successivamente richiesta un'operazione di ordinamento inevitabile. Ovviamente i minori tempi di esecuzione sono ottenuti solamente quando sono considerate relazioni di piccole dimensioni, in quanto PostgreSQL non dispone di tecniche simili a SQL Server.
Anche per Oracle XE vale quanto appena detto per PostgreSQL, infatti tutti i piani di esecuzione presentati risultano basati sulla stessa identica struttura, senza differenze sostanziali. Si nota inoltre che l'unica tipologia di join realizzata da Oracle XE è rappresentata dagli Hash Join, quindi, mentre da una parte PostgreSQL sfrutta la riduzione di righe apportata dall'imposizione di condizioni restrittive che gettano le basi per i benefici di un Nested Loop indicizzato, dall'altra Oracle XE mantiene costantemente l'algoritmo di Hash Join.
Strategia analoga adottata anche da parte di MySQL, che in tutti i piani di esecuzione ricavati nel corso dello studio ha mantenuto la medesima struttura, basata sull'uso esclusivo dell'algoritmo di Merge Join, unico supportato.
La strategia vincente risulta quella adottata da SQL Server e PostgreSQL, che sono in grado, più degli altri DBMS, di identificare le situazioni differenti in cui un algoritmo di join si qualifica come più adatto ed efficiente rispetto gli altri.

In merito al terzo punto, si può notare come né SQL Server, né PostgreSQL e né Oracle XE siano in grado, a partire da query immesse così impostate, di capire quale possa essere l'ordine più conveniente in base ai quali eseguire i Full Outer Join. Infatti esprimendo i Full Outer Join in questo modo, coinvolgendo tre relazioni, essi vengono sempre tradotti in due diversi join, uno realizzato tra le prime due tabelle inserite nella clausola FROM e il secondo tra il risultato del primo e la terza relazione rimanente.
Per PostgreSQL, Oracle XE e SQL Server risulta del tutto indifferente se le prime due tabelle che compaiono nella clausola FROM vengono scambiate tra loro nella query: in SQL Server queste vengono unite mediante Merge Join, pertanto è ininfluente il loro ordine, e nel caso di PostgreSQL e Oracle XE sono invece unite tramite Hash Join pertanto, indipendentemente dall'ordine nel testo della query, viene sempre eletta come build input quella di dimensioni minori e come probe input quella di dimensioni maggiori.
Le prestazioni cambiano scambiando tra loro la prima e la terza relazione nella clausola FROM, riscontrando tempi di esecuzione inferiori qualora fosse inserita dapprima la tabella di cardinalità minore.
Se ci si sofferma in particolare sulle query in cui non è imposta alcuna condizione, si può notare che ciò che i tre DBMS citati non sono in grado di realizzare è il seguente compito: avendo a disposizione tre relazioni che devono essere tutte e tre unite mediante Full Outer Join, determinare quale può essere la combinazione nel query plan più conveniente in termini di costo che permetta di ottenere il risultato cercato.
Per MySQL non possono essere valide le medesime considerazioni, in quanto i Full Outer Join

possono essere solamente simulati; si può tuttavia osservare che l'ordine secondo il quale le relazioni coinvolte nei join vengono introdotte nei piani di esecuzione corrisponde a quello secondo il quale sono inserite le tabelle nella clausola FROM dell'interrogazione immessa. È prevista solo una leggera ottimizzazione che prevede di coinvolgere per prima, quando possibile, la tabella di cardinalità inferiore, sulla quale è eseguita un'operazione di Full Table Scan, più onerosa per tabelle di elevate dimensioni.
In ultima analisi quindi anche MySQL tende a non determinare sempre l'ordine ottimale degli outer join (left/right), ma rispetto ai tre DBMS citati in precedenza, attua l'ottimizzazione detta.
Ovviamente è importante ribadire che in questo contesto le osservazioni sono valide esclusivamente per le operazioni di Full Outer Join, e non è quindi lecito estendere tali considerazioni anche per le operazioni di Inner Join, per le quali l'optimizer cambia adeguatamente l'ordine in fase di ottimizzazione.
Si sottolinea, tuttavia, che l'optimizer, per realizzare tale compito, cerca di individuare quale ordine di join fornisce il resultset più piccolo, ma per query complesse non può indagare tutte le possibili combinazioni, quindi anche l'ordine con cui compaiono le relazioni tra cui eseguire Inner join risulta in ultima analisi influente sulle prestazioni, anche se in maniera minore rispetto gli operatori di Full Outer Join [14].

Si riportano ora i principali dati tratti dalle analisi precedentemente riportate.

GRUPPO 1

Tempi di esecuzione	*SQL Server*	*PostgreSQL*	*Oracle*	*MySQL*	*Righe ritornate*
Query 1 / senza COALESCE	00:01:18 / 00:02:55	00:11:37	00:37:39 / 00:43:49	26:13:00	6,599,143
Query 2 (1 condizione)	00:00:49	00:07:29	00:45:42	00:38:15	4,184,487
Query 3 (2 condizioni)	00:00:06	00:01:32	00:02:01	00:01:28	481,357
Query 4 (3 condizioni)	00:00:03	00:00:34.5	00:01:22	00:00:12	110,753

Tabella 12

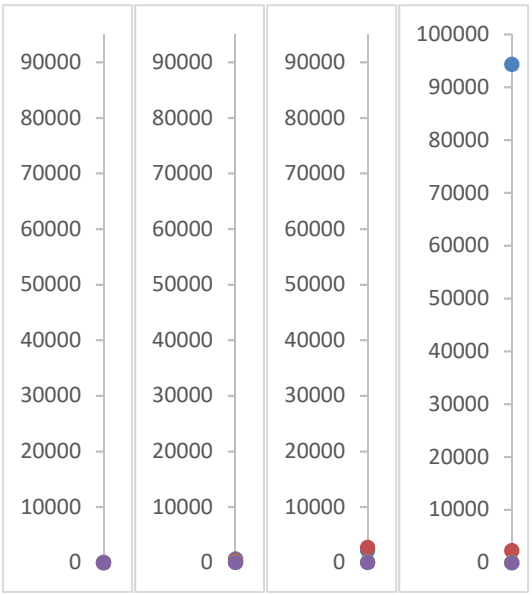


Tabella 13 - Distribuzione dei tempi di esecuzione.
Da sinistra a destra: SQL Server, PostgreSQL, Oracle XE e MySQL

SQL Server svetta come il più veloce, registrando tempi di esecuzione in media dell'89.3% superiori rispetto PostgreSQL, del 97.28% rispetto Oracle XE e del 99.86% rispetto MySQL. Oracle XE e MySQL si contendono le prestazioni peggiori: MySQL, infatti, per la prima query registra un tempo di esecuzione assolutamente non competitivo, mentre mano a mano che vengono imposte delle condizioni, e quindi scremate le righe da processare, le sue performance aumentano, tanto da renderlo il secondo DBMS più veloce per la Query 3 e la Query 4, scavalcando PostgreSQL. Oracle invece si conferma come più lento per Query 2, Query 3 e Query 4, registrando, in relazione alle query citate, delle prestazioni inferiori di 98 % rispetto SQL Server, 80.46% rispetto PostgreSQL e 18.68% rispetto MySQL.

Riformulazione per compatibilità con MySQL:

Tempi di esecuzione	*SQL Server*	*PostgreSQL*	*Oracle*	*MySQL*
Query 1 (no condizione) / Con vista intermedia	00:01:35 / 00:02:34	00:17:46 / 00:09:54	00:18:44/ 00:20:03	*26:13:00*/ 00:03:58
Query 2 (1 condizione)	00:00:52	00:10:12	00:19:43	00:38:15
Query 3 (2 condizioni)	00:00:09	00:02:27	00:05:32	00:01:28
Query 4 (3 condizioni)	00:00:03	00:00:53.6	00:03:44	00:00:12

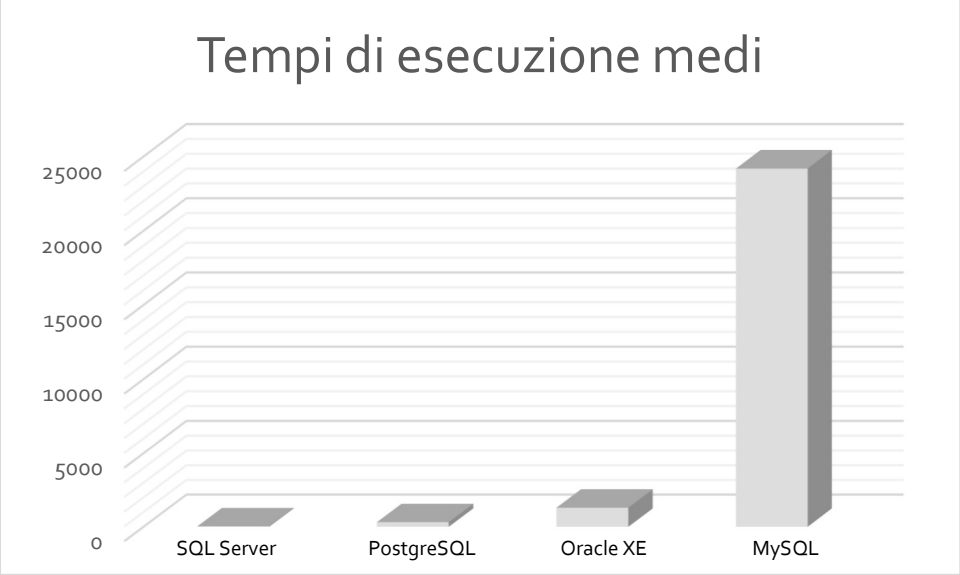
Tabella 14

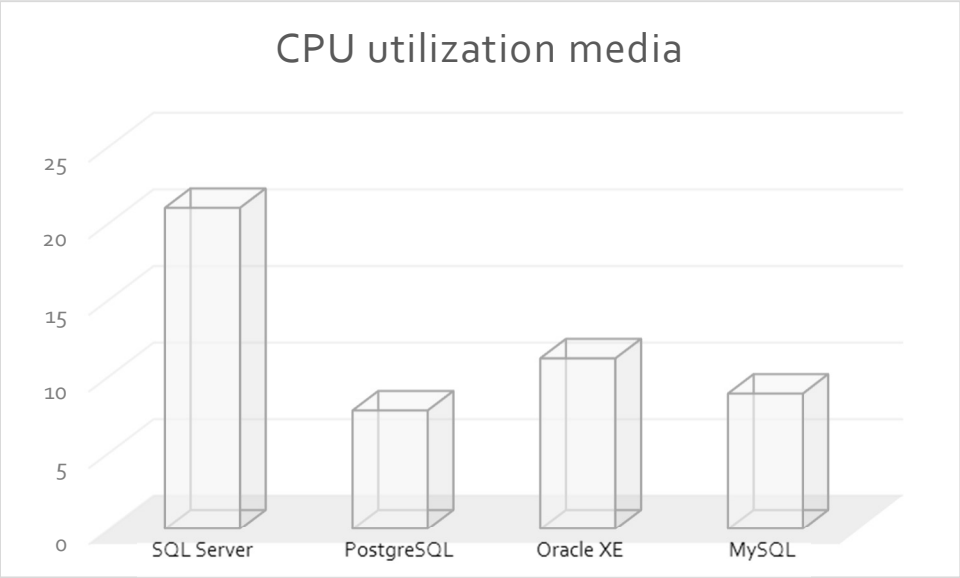
SQL Server e PostgreSQL eseguono tutte le query con tempi superiori utilizzando la formulazione di MySQL, rispetto la versione precedente. Si tratta di un peggioramento medio pari a 19.48% per SQL Server e pari a 48.99% per PostgreSQL.
Oracle XE invece trae giovamento dalla nuova formulazione quando le righe ritornate non sono scremate da più di una condizione, quando quindi vi è un elevato numero di righe che le operazioni devono elaborare; in tali circostanze i dati dimostrano che Oracle XE raggiunge un

incremento medio delle performance del 56.9%; invece in presenza di almeno due condizioni che permettono una diminuzione delle righe più consistente, i tempi di esecuzione nella formulazione che prevede Full Outer Join si abbassano drasticamente e non risultano più convenienti i piani di esecuzione prodotti con la seconda formulazione.
SQL Server si riconferma il più veloce, mentre Oracle il più lento per le query 3 e 4, MySQL registra invece i tempi peggiori relativamente alla query 1 e 2; in assenza di condizioni o con l'applicazione di una singola condizione non selettiva, quindi MySQL è il più lento, mentre quando le righe da processare non sono elevate, perché imposti più filtri, MySQL si classifica come secondo più veloce.
Nella formulazione basata sull'utilizzo di una vista intermedia rimane ancora SQL Server il più veloce, dovendo probabilmente tale risultato all'impiego del parallelismo.
MySQL risulta invece il secondo più veloce, raggiungendo tale posizione grazie alla tabella temporanea sfruttata per non dover ricreare i risultati della vista intermedia, che ha favorito la diminuzione del tempo impiegato per l'esecuzione della query. Rimane invece più conveniente la scelta operata da SQL Server, Oracle XE e PostgreSQL di realizzare la vista sfruttando un join in meno rispetto quelli impiegati da MySQL.

	Tempo di esecuzione medio	*CPU utilization media*	*Numero di threads impiegati*
SQL Server	34 secondi	21.02%	8
PostgreSQL	318.125 secondi	7.8%	1
Oracle XE	1301 secondi	11.2%	1
MySQL	24193.75 secondi	8.9%	1

Tabella 15





GRUPPO 2

Tempi di esecuzione	*SQL Server*	*PostgreSQL*	*Oracle*	*MySQL*	*Righe ritornate*
Query 1 / senza COALESCE	00:00:36 / 00:01:07	00:04:58	01:17:00/ 26:02:00	25:34:00	*2,901,770*
Query 2 (1 condizione)	00:00:27	00:03:17	00:15:33	00:25:51	1,982,047
Query 3 (2 condizioni)	00:00:02.8	00:00:29.2	00:01:24	00:02:01	184,359
Query 4 (2 condizioni sel.)	00:00:02.3	00:00:16.7	00:02:54	00:01:10	108,379
Query 5 (3 condizioni)	00:00:01	00:00:5.4	00:01:13	00:00:12	20,086

Tabella 16

SQL Server mantiene il vantaggio sugli altri DBMS, registrando tempi di esecuzione inferiori del 87.35% rispetto PostgreSQL, del 98.8% rispetto Oracle XE e del 99% rispetto MySQL.
Anche in questo caso il tempo di esecuzione impiegato da MySQL per la Query 1 non è competitivo rispetto gli altri tre DBMS. Come nel caso precedente, infatti, MySQL ottiene prestazioni nettamente superiori solo quando le righe da elaborare non sono elevate, altrimenti la strategia scelta risulta per nulla efficiente. Differentemente dal gruppo 1, MySQL si conferma per ogni query del gruppo ora esaminato come il secondo peggiore, dopo Oracle XE, per le Query 4 e Query 5, mentre il meno efficiente per la Query 1,2 e3.
In questo caso quindi PostgreSQL mantiene la seconda posizione in relazione ad ogni query.

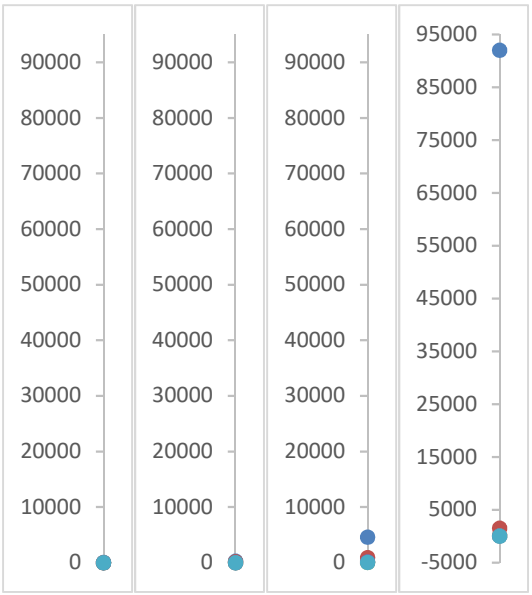


Tabella 17 - Distribuzione dei tempi di esecuzione.
Da sinistra a destra: SQL Server, PostgreSQL, Oracle XE e MySQL

Riformulazione per compatibilità con MySQL:

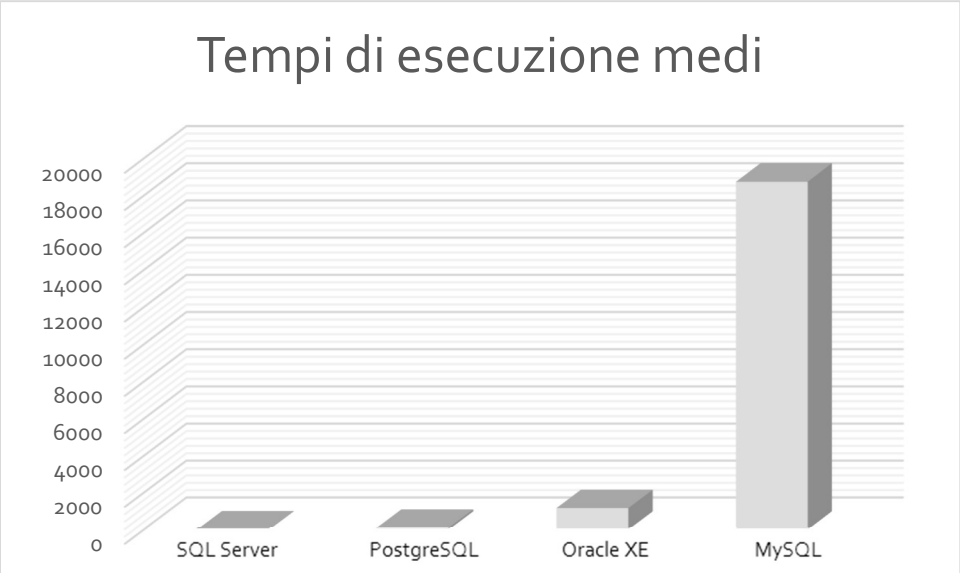
Tempi di esecuzione	*SQL Server*	*PostgreSQL*	*Oracle*	*MySQL*
Query 1 (no condizione)	00:00:40	00:05:20	00:36:51	25:34:00
Query 2 (1 condizione)	00:00:30	00:03:46	00:07:20	00:25:51
Query 3 (2 condizioni)	00:00:04	00:00:31.6	00:02:22	00:02:01
Query 4 (2 condizioni sel.)	00:00:04	00:00:27.3	00:02:19	00:01:10
Query 5 (3 condizioni)	00:00:01.902	00:00:05.7	00:01:14	00:00:12

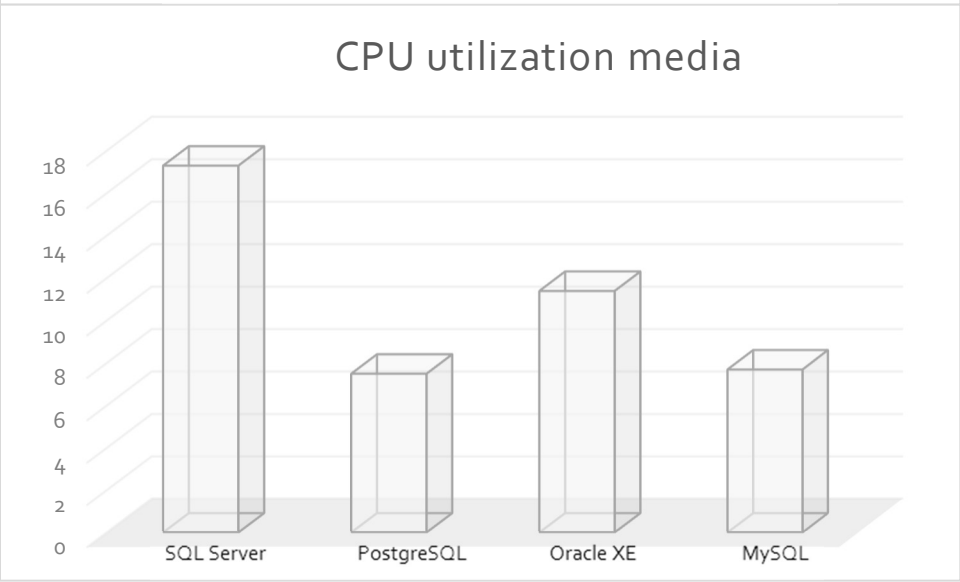
Tabella 18

Anche in questo caso SQL Server e PostgreSQL vedono i propri tempi di esecuzione peggiorati in questa formulazione di un valore medio pari a 13.52% per SQL Server e pari a 10.53% per PostgreSQL.
Al contrario, Oracle XE registra tempi di esecuzione migliori, con un miglioramento medio pari al 48.91%.

	Tempo di esecuzione medio	*CPU utilization media*	*Numero di threads impiegati*
SQL Server	13.82 secondi	17.3%	8
PostgreSQL	109.26 secondi	7.5%	1
Oracle XE	1176.8 secondi	11.4%	1
MySQL	18758.8 secondi	7.7%	1

Tabella 19





GRUPPO 3

Tempi di esecuzione	*SQL Server*	*PostgreSQL*	*Oracle*	*MySQL*	*Righe ritornate*
Query 1 (no condizione)	00:00:19	00:02:28	00:02:40	00:07:31	*1,574,218*
Query 2 (1 condizione)	00:00:10	00:01:19	00:01:30	00:04:55	812,639
Query 3 (1 condizione sel.)	00:00:00.503	00:00:04.6	00:00:20.1	00:00:05.34	16,395
Query 4 (2 condizioni)	00:00:00.437	00:00:03.2	00:00:47.74	00:00:06.4	19,145
Query 5 (3 condizioni)	00:00:00.536	00:00:00.842	00:00:21.45	00:00:03.8	134

Tabella 20

Rimangono valide le osservazioni presentate in merito alle tabelle precedenti: SQL Server risulta il DBMS più veloce, con tempi di esecuzione inferiori del 87.06% rispetto PostgreSQL, 91 % rispetto Oracle XE e 95.99% rispetto MySQL.
PostgreSQL invece si posiziona al secondo posto, in relazione a tutte le query presentate, mentre,

ancora una volta, le prestazioni peggiori sono quelle di MySQL in relazione alle prime due query, mentre di Oracle XE in relazione alle ultime tre; questo perché, proprio come accadeva precedentemente, MySQL ottiene risultati prestazionali considerevolmente migliori con poche righe da elaborare.

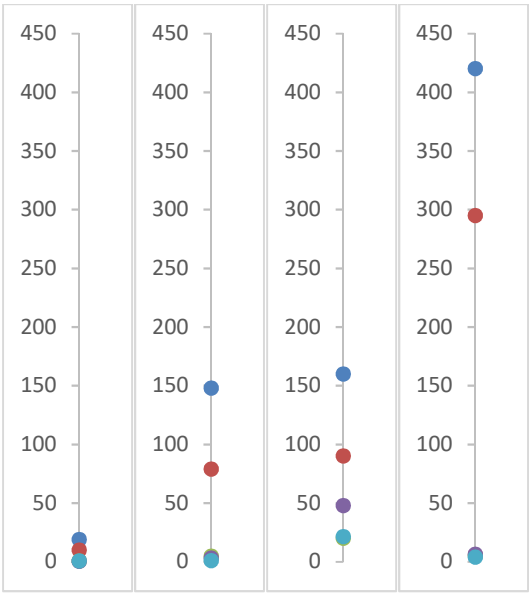


Tabella 21 - Distribuzione dei tempi di esecuzione.
Da sinistra a destra: SQL Server, PostgreSQL, Oracle XE e MySQL

Riformulazione per compatibilità con MySQL:

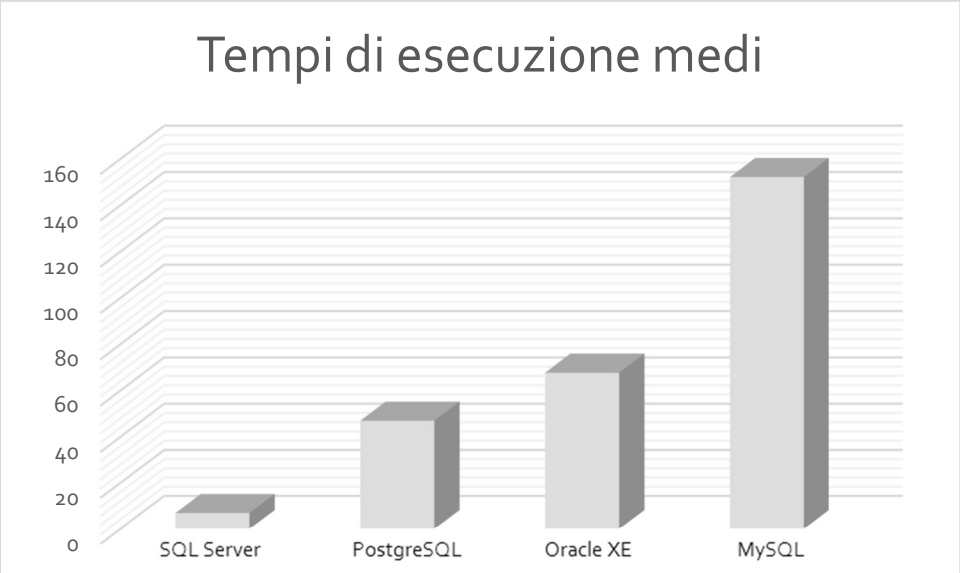
Tempi di esecuzione	*SQL Server*	*PostgreSQL*	*Oracle*	*MySQL*
Query 1 (no condizione)	00:00:24	00:02:52	00:04:21	00:07:31
Query 2 (1 condizione)	00:00:13	00:01:33	00:03:25	00:04:55
Query 3 (1 condizione sel.)	00:00:01.053	00:00:02.5	00:00:48.86	00:00:05.34
Query 4 (2 condizioni)	00:00:01.384	00:00:04.3	00:00:37	00:00:06.44
Query 5 (3 condizioni)	00:00:00.439	00:00:00.776	00:00:12.482	00:00:03.8

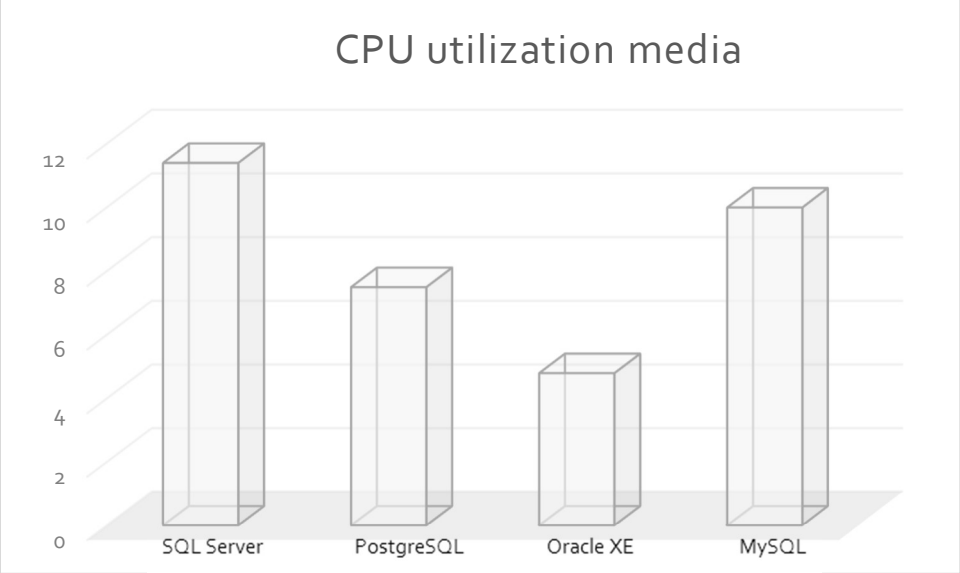
Tabella 22

SQL Server registra un peggioramento medio pari al 23.57%, così come PostgreSQL vede i propri tempi di esecuzione ridotti del 13.55%.
Oracle XE stavolta ottiene prestazioni migliori con questa formulazione solo relativamente alle Query 4 e 5, subendo un miglioramento medio pari al 28.5%, considerando solo le ultime due query, mentre considerando i tempi di esecuzione globalmente registrati, si verifica un peggioramento complessivo del 39.87%.

	Tempo di esecuzione medio	*CPU utilization media*	*Numero di threads impiegati*
SQL Server	6.095 secondi	11.4%	8
PostgreSQL	47.13 secondi	4.8%	1
Oracle XE	67.86 secondi	10%	1
MySQL	152.316 secondi	6.8%	1

Tabella 23





GRUPPO 4

Tempi di esecuzione	*SQL Server*	*PostgreSQL*	*Oracle*	*MySQL*	*Righe ritornate*
Query 1	00:00:09	00:01:13	00:01:16	00:01:52	*803,290*
Query 2 (1 condizione)	00:00:02.33	00:00:17	00:00:23.98	00:00:14.72	188,491
Query 3 (2 condizioni)	00:00:00.202	00:00:00.72	00:00:07.387	00:00:03.5	2,163
Query 4 (3 condizioni)	00:00:00.143	00:00:00.57	00:00:06.187	00:00:02.15	26

Tabella 24

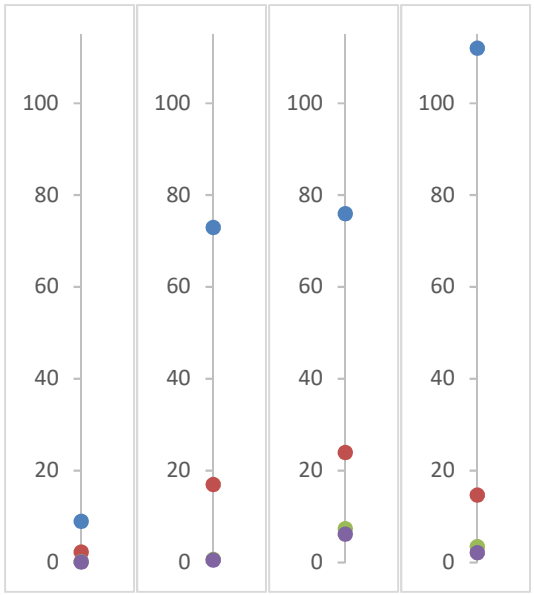


Tabella 25 - Distribuzione dei tempi di esecuzione.
Da sinistra a destra: SQL Server, PostgreSQL, Oracle XE e MySQL

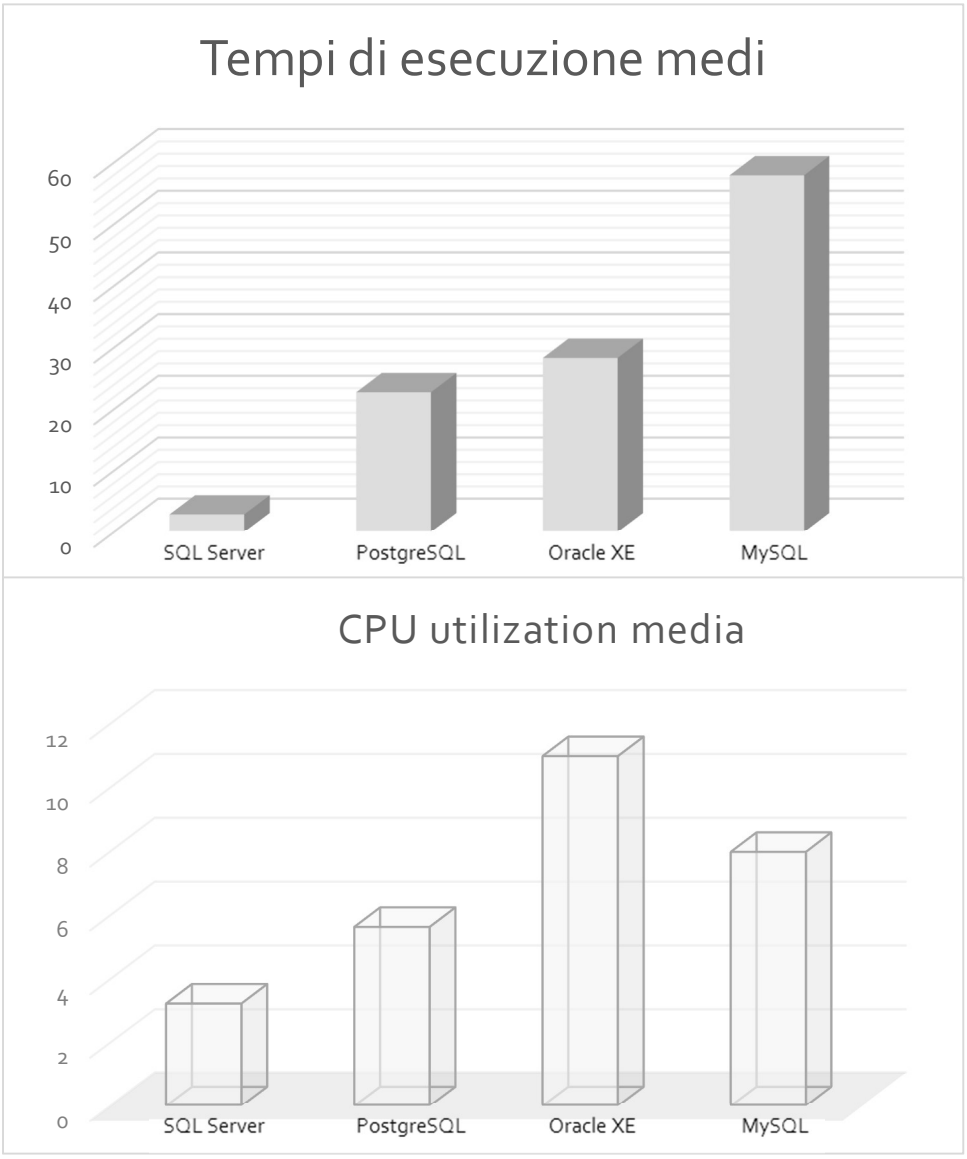
Riformulazione per compatibilità con MySQL:

Tempi di esecuzione	*SQL Server*	*PostgreSQL*	*Oracle*	*MySQL*
Query 1 (no condizione) /	00:00:13	00:01:21	00:02:02	00:01:52
Con vista intermedia	00:00:09	00:01:25	00:02:52	00:01:44
Query 2 (1 condizione)	00:00:02	00:00:19.3	00:00:46	00:00:14.72
Query 3 (2 condizioni)	00:00:00.592	00:00:01.1	00:00:14.69	00:00:03.5
Query 4 (3 condizioni)	00:00:00.172	00:00:00.336	00:00:05.974	00:00:02.156

Tabella 26

	Tempo di esecuzione medio	*CPU utilization media*	*Numero di threads impiegati*
SQL Server	2.91 secondi	3.25%	8
PostgreSQL	22.82 secondi	5.65%	1
Oracle XE	28.39 secondi	11%	1
MySQL	58.094 secondi	8%	1

Tabella 27



SQL Server si conferma, per ogni tabella riportata, il più performante.
MySQL si conferma essere, in media, il DBMS con prestazioni più scarse; infatti, nonostante in caso di tabelle di ridotte dimensioni abbia dei tempi di esecuzione competitivi con gli altri DBMS, i tempi registrati per l'elaborazione di tabelle di elevata cardinalità sono notevolmente superiori, alzando di conseguenza i valori medi registrati.

I valori relativi all'utilizzo di CPU si mantengono pressoché costanti per PostgreSQL, Oracle XE e MySQL, contestualmente ai quattro gruppi; tra i tre DBMS, Oracle XE è quello che registra valori leggermente superiori di CPU utilization.
Per SQL Server i valori variano con intensità maggiore. Infatti, essendo l'unico a ricorrere alla tecnica del parallelismo, presenta situazioni con un ridotto consumo di CPU laddove vengano prediletti piani seriali, mentre nel caso di piani paralleli si riscontrano valori di CPU utilization più elevati. Si sottolinea che quando il query optimizer di SQL Server opta per piani seriali, i valori di

utilizzo di CPU si mantengono inferiori rispetto quelli presentati nelle medesime condizioni dagli altri tre DBMS.

Si riportano ora i dati dei ranking basati sui due criteri principali impiegati nello studio: tempo di esecuzione e utilizzo di CPU, considerati nei valori medi globali in relazione alla totalità dei gruppi. I dati sono stati normalizzati applicando il rapporto tra il singolo valore e il valore massimo .

CPU: 0.5
Tempo: 0.5

	Criteri			
DBMS	CPU	Tempo	Finale	Rango
SQL Server	1	0.001316	0.500658	3
PostgreSQL	0.486405	0.011523	0.248964	1
Oracle XE	0.823263	0.059639	0.441451	2
MySQL	0.5929	1	0.79645	4

CPU: 0
Tempo: 1

	Criteri			
DBMS	CPU	Tempo	Finale	Rango
SQL Server	1	0.001316	0.001316	1
PostgreSQL	0.486405	0.011523	0.011523	2
Oracle XE	0.823263	0.059639	0.059639	3
MySQL	0.5929	1	1	4

CPU: 1
Tempo: 0

	Criteri			
DBMS	CPU	Tempo	Finale	Rango
SQL Server	1	0.001316	1	4
PostgreSQL	0.486405	0.011523	0.486405	1
Oracle XE	0.823263	0.059639	0.823263	3
MySQL	0.5929	1	0.5929	2

Le tre tabelle sopra riportate mostrano che i DBMS più performanti sono diversi a seconda dell'importanza attribuita ai criteri utilizzati per l'indagine: nel caso in cui risulti prevalente l'importanza associata al tempo di esecuzione, trascurando i dati di utilizzo di CPU, il DBMS più indicato è MS SQL Server; qualora, invece, sia preponderante il criterio di utilizzo di CPU, si configura come DBMS più efficiente PostgreSQL. Infine, se i due criteri rivestono la medesima importanza, PostgreSQL si conferma il più indicato, mentre SQL Server riveste solamente la terza posizione.

In relazione agli indici, essi portano ad un incremento considerevole nei tempi di esecuzione, rendendo le prestazioni ovviamente migliori.
I dati comparativi in assenza e presenza di indici sono indicati nelle tabelle riportate nel capitolo precedente; vengono ora riassunti i risultati tramite i **valori medi degli incrementi di prestazioni**:

SQL SERVER	POSTGRESQL	ORACLE XE	MYSQL
64.6%	53.5%	13.5%	67.7%

I dati sopra riportati dimostrano che sono MySQL e SQL Server i DBMS che traggono maggior giovamento dall'impiego di indici, appositamente creati sui campi interessati dall'applicazione di condizioni di filtro; mentre Oracle XE è il DBMS che, al contrario, beneficia in misura minore di tali indici, ottenendo prestazioni di poco superiori rispetto al loro mancato impiego.

Ovviamente la lettura dei dati sopra riportati deve avvenire mantenendo sempre chiara la distinzione tra MS SQL Server da una parte, e PostgreSQL, Oracle XE e MySQL dall'altra, in quanto si ribadisce che, essendo il primo l'unico disponibile con licenza a pagamento, è lecito e comprensibile che le sue prestazioni siano nettamente superiori rispetto a quelle registrate nell'ambito degli altri tre DBMS coinvolti. Si precisa ulteriormente, inoltre, che i dati ricavati in merito allo studio svolto riguardano solamente test comparativi incentrati su operazioni di Full Outer Join, pertanto, sebbene tali informazioni siano indicative per stabilire un confronto tra i DBMS presentati, non sono esaustive e non coinvolgono operazioni e aspetti più globali.
Al fine di fornire al lettore ulteriori informazioni relative al confronto prestazionale, con le quali integrare i dati riportati nella presente tesi, si riporta uno schema, tratto da DB Engines [15], che offre una visione di più ampio spettro, coinvolgendo anche altri DBMS, a pagamento e non. Lo schema inoltre evidenzia come non sia possibile stilare una classifica assoluta dei DBMS in commercio, in quanto essi possono essere studiati ed indagati impiegando criteri differenti e adottando diversi approcci, con il risultato di produrre classifiche anche molto diverse tra loro. Nel caso ora esaminato, per esempio, il criterio di ranking è basato su un aspetto finora nemmeno menzionato: la popolarità dei DBMS, misurata tramite i seguenti parametri:

1. Numero di menzioni del sistema sui siti web, misurato come numero di risultati nelle query dei motori di ricerca (quelli impiegati sono Google, Bing e Yandex).
2. Interesse generale verso il sistema, usando la frequenza delle ricerche di Google Trends.
3. Frequenza di discussioni tecniche riguardanti i DBMS, sui siti Q&A Stack Overflow e DBA Stack Exchange.
4. Numero di offerte di lavoro, in cui il sistema è menzionato.
5. Numero di profili nelle reti professionali, in cui il sistema è citato (LinkedIn e Upwork).
6. Rilevanza nei social networks: è contato il numero di tweet[18] nei quali il sistema è menzionato.

[18] Messaggi di testo, di lunghezza massima di 140 caratteri, impiegati dagli utenti di Twitter.

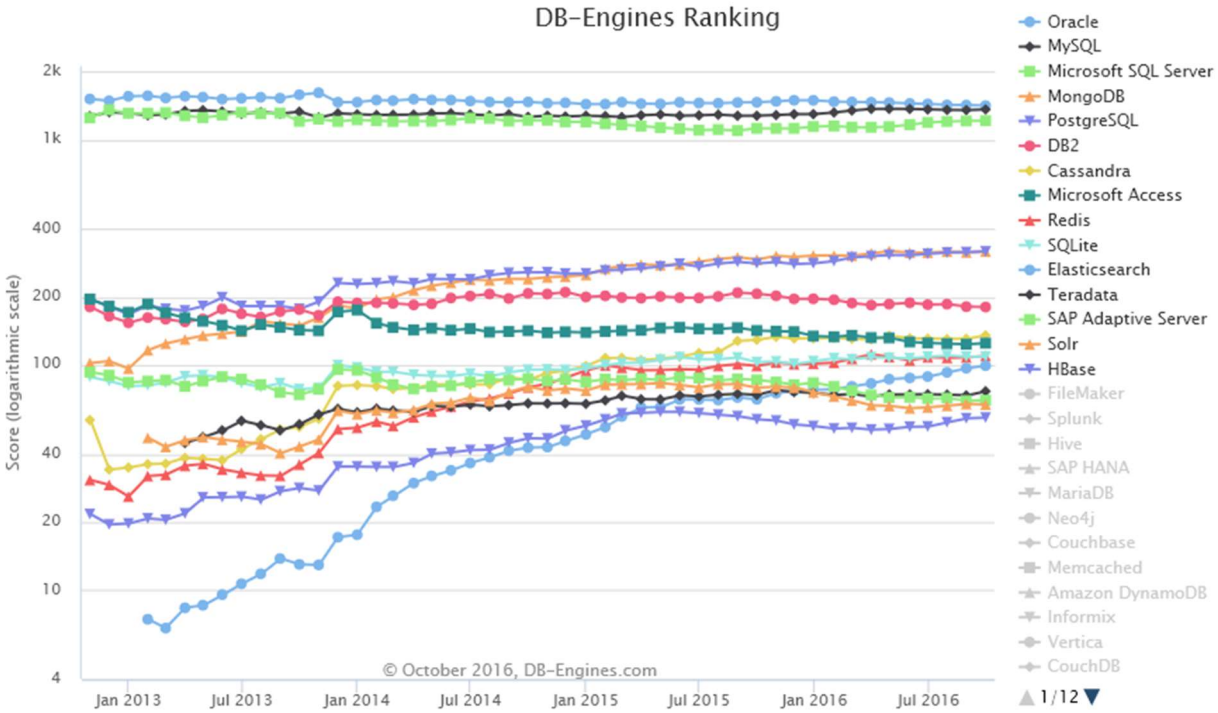

DB-Engines Ranking
Score (logarithmic scale)
2k
1k
400
200
100
40
20
10
4
Jan 2013
Jul 2013
Jan 2014
Jul 2014
Jan 2015
Jul 2015
Jan 2016
Jul 2016
© October 2016, DB-Engines.com
Oracle
MySQL
Microsoft SQL Server
MongoDB
PostgreSQL
DB2
Cassandra
Microsoft Access
Redis
SQLite
Elasticsearch
Teradata
SAP Adaptive Server
Solr
HBase
FileMaker
Splunk
Hive
SAP HANA
MariaDB
Neo4j
Couchbase
Memcached
Amazon DynamoDB
Informix
Vertica
CouchDB
1/12

Figura 68

Infine, si può così riassumere il confronto:

	SQL Server	*PostgreSQL*	*MySQL*	*Oracle*
Supporto Full Outer Join	✓	✓	✗	✓
Supporto per diverse sintassi	✓	✗	✗	✓
Join Supportati	Merge Join, Hash Join, Nested Loop Join	Merge Join, Hash Join, Nested Loop Join	Nested Loop Join	Merge Join, Hash Join, Nested Loop Join. Preferisce sempre Hash Join.
Parallelismo	✓	✗	✗	✗
Indice sulla chiave primaria	Clustered	Non clustered	Clustered	Non clustered
Table spool	✓	✗	✗	No, ma utilizza viste intermedie
Index Seek dinamico	✓	✗	✗	✓
Merge Interval	✓	✗	✗	✗
Filtri bitmap	✓	✗	✗	✗
Filter	✓	✓	✗	✓
Individuazione ordine ottimale	✗	✗	✗	✗

Tabella 28

APPENDICE A – SCRIPT PER LA CREAZIONE DELLE TABELLE

```
CREATE TABLE flight_1_1 AS

      SELECT *
      FROM flight_1 f1
      WHERE f1.distance='NA'
      OR f1.distance='Distance'
      OR 350>(SELECT NULLIF(f1.distance, '')::int)

ALTER TABLE flight_1_1 ADD PRIMARY KEY (id)


CREATE TABLE flight_1_2 AS

  SELECT *
  FROM flight_1
  WHERE origin = 'ATL' or dest = 'ATL'

ALTER TABLE flight_1_2 ADD PRIMARY KEY (id)


CREATE TABLE flight_1_3 AS

      SELECT *
      FROM flight_1
      WHERE uniquecarrier='B6'

ALTER TABLE flight_1_3 ADD PRIMARY KEY (id)


CREATE TABLE flight_2_1 AS

      SELECT *
      FROM flight_2 f2
      WHERE (f2.crsdeptime='CRSDepTime'
      OR 920 < (SELECT NULLIF(f2.crsdeptime, '')::int)  )

      AND (f2.crsarrtime='CRSArrTime'
      OR 1200>(SELECT NULLIF(f2.crsarrtime, '')::int))

ALTER TABLE flight_2_1 ADD PRIMARY KEY (id)


CREATE TABLE flight_2_2 AS

      SELECT *
      FROM flight_2
      WHERE deptime = 'NA';

ALTER TABLE flight_2_2 ADD PRIMARY KEY (id)


CREATE TABLE flight_2_3 AS

      SELECT *
      FROM flight_2 f2
      WHERE f2.crsdeptime='CRSDepTime'
      OR 700 < (SELECT NULLIF(f2.crsdeptime, '')::int)

ALTER TABLE flight_2_3 ADD PRIMARY KEY (id)
```

```
CREATE TABLE flight_3_1 AS

  SELECT * FROM flight_3 WHERE cancelled = '1';

ALTER TABLE flight_3_1 ADD PRIMARY KEY (id);


CREATE TABLE flight_3_2 AS

      SELECT * FROM flight_3
      WHERE (carrierdelay<>'NA' and 60 < (SELECT NULLIF(carrierdelay, '')::int))
      OR (weatherdelay<>'NA' and 60 < (SELECT NULLIF(weatherdelay, '')::int))
      OR (nasdelay<>'NA' and 60<(SELECT NULLIF(nasdelay, '')::int))
      OR (securitydelay<>'NA' and 60<(SELECT NULLIF(securitydelay, '')::int))
      OR (lateaircraftdelay<>'NA' and 60< (SELECT NULLIF(lateaircraftdelay,
      '')::int));

ALTER TABLE flight_3_2 ADD PRIMARY KEY (id)
```

APPENDICE B – QUERY IN SUPPORTO ALLA MIGRAZIONE DEL DB

MS SQL SERVER

```
SELECT *
INTO flight_2_1
FROM flight_2 f2
WHERE (f2.crsdeptime='CRSDepTime'
OR 920 < (SELECT CAST(f2.crsdeptime AS INT) )
      )
AND (f2.crsarrtime='CRSArrTime'
OR 1200>(SELECT CAST(f2.crsarrtime AS INT) )
      )


SELECT *
INTO flight_2_3
FROM flight_2 f2
WHERE (f2.crsdeptime='CRSDepTime'
OR 700 < (SELECT NULLIF (CAST(f2.crsdeptime AS INT),''  ) )
      )
```

MYSQL

```
CREATE TABLE flight_2_1 AS
SELECT *
FROM flight_2 f2
WHERE (f2.crsdeptime='CRSDepTime'
OR 920 < (SELECT CAST(f2.crsdeptime AS UNSIGNED) )
      )
AND (f2.crsarrtime='CRSArrTime'
OR 1200>(SELECT CAST(f2.crsarrtime AS UNSIGNED) )
      )


CREATE TABLE flight_2_3 AS
```

```
SELECT *
FROM flight_2 f2
WHERE (f2.crsdeptime='CRSDepTime'
OR 700 < (SELECT NULLIF (CAST(f2.crsdeptime AS UNSIGNED),''  ) )
      )
```

ORACLE XE

```
CREATE TABLE flight_2_1 AS (SELECT *
                            FROM flight_2 f2
                            WHERE 920 <(SELECT CAST(f2b.crsdeptime AS NUMBER)
                                        FROM flight_2 f2b
                                        WHERE f2b.id=f2.id)

                            AND 1200> (SELECT CAST(f2b.crsarrtime AS NUMBER)
                                        FROM flight_2 f2b
                                        WHERE f2b.id=f2.id)

                            )

CREATE TABLE flight_2_3 AS (SELECT *
                            FROM flight_2 f2
                            WHERE 700 < (SELECT CAST(f2b.crsdeptime AS NUMBER)
                                         FROM flight_2 f2b
                                         WHERE f2b.id=f2.id)

                            )
```

9 BIBLIOGRAFIA

[1] S. Bergamaschi, Introduzione ai DBMS.

[2] Y. E. Ioannidis, «Query Optimization,» [Online]. Available: http://cs.stanford.edu/people/widom/cs346/ioannidis.pdf.

[3] MS TechNet, «Utilizzo di outer join,» Microsoft, [Online]. Available: https://technet.microsoft.com/it-it/library/ms187518(v=sql.105).aspx.

[4] MS TechNet, «Mirroring del database (SQL Server),» MS TechNet, [Online]. Available: https://msdn.microsoft.com/it-it/library/ms189852.aspx.

[5] SQL Articles, [Online]. Available: http://sql-articles.com/.

[6] MS TechNet, «Clustered and Nonclustered Indexes Described,» [Online]. Available: https://msdn.microsoft.com/it-it/library/ms190457(v=sql.120).aspx.

[7] StackOverflow, «Differences between a clustered and a non-clustered index,» [Online]. Available: http://stackoverflow.com/questions/91688/what-are-the-differences-between-a-clustered-and-a-non-clustered-index.

[8] B. Ozar, «What is the CXPACKET Wait Type, and How Do You Reduce It?,» [Online]. Available: https://www.brentozar.com/archive/2013/08/what-is-the-cxpacket-wait-type-and-how-do-you-reduce-it/.

[9] P. White, «Parallel Execution Plans – Branches and Threads,» 7 Ottobre 2013. [Online]. Available: https://sqlperformance.com/2013/10/sql-plan/parallel-plans-branches-threads.

[10] TutorialsPoint, «SQL - Indexes,» [Online]. Available: http://www.tutorialspoint.com/sql/sql-indexes.htm.

[11] PostrgreSQL Manuals, «Performance Tips,» [Online]. Available: https://www.postgresql.org/docs/8.2/static/using-explain.html.

[12] MySQL Reference Manual, «Range Optimization,» [Online]. Available: http://dev.mysql.com/doc/refman/5.7/en/range-optimization.html.

[13] Use the Index, Luke, «Index-Only Scan: Avoiding Table Access,» [Online]. Available: http://use-the-index-luke.com/sql/clustering/index-only-scan-covering-index.

[14] B. Snaidero, «How Join Order Can Affect the Query Plan,» [Online]. Available: https://www.mssqltips.com/sqlservertutorial/3201/how-join-order-can-affect-the-query-plan/.

[15] DB-Engines, «DB-Engines Ranking - Trend Popularity,» [Online]. Available: http://db-engines.com/en/ranking_trend.

[16] A. Bansal, «SQL Server Cost Threshold for Parallelism,» 29 Maggio 2012. [Online]. Available: http://www.sqlservergeeks.com/sql-server-cost-threshold-for-parallelism/.

10 RINGRAZIAMENTI

Il primo grazie va indubbiamente ai miei genitori, senza il supporto dei quali non avrei raggiunto questo obiettivo, perché mi hanno sempre messa nelle condizioni migliori per affrontare al meglio il mio lavoro e perché mi hanno sempre sostenuta nelle mie scelte.

Ma il ringraziamento a cui tengo maggiormente è per Stefano, per il supporto, l'aiuto e la comprensione sempre dimostrata. Grazie per essere la mia coperta di Linus, per le conversazioni stimolanti, per la sana competizione che mi ha sempre spronata a fare del mio meglio, grazie per essere un esempio cui fare riferimento e infine grazie per essere la mia cheerleader personale, perché so che in te troverò sempre appoggio e tifo, che non mi hai mai fatto mancare nei sette anni in cui siamo cresciuti insieme.